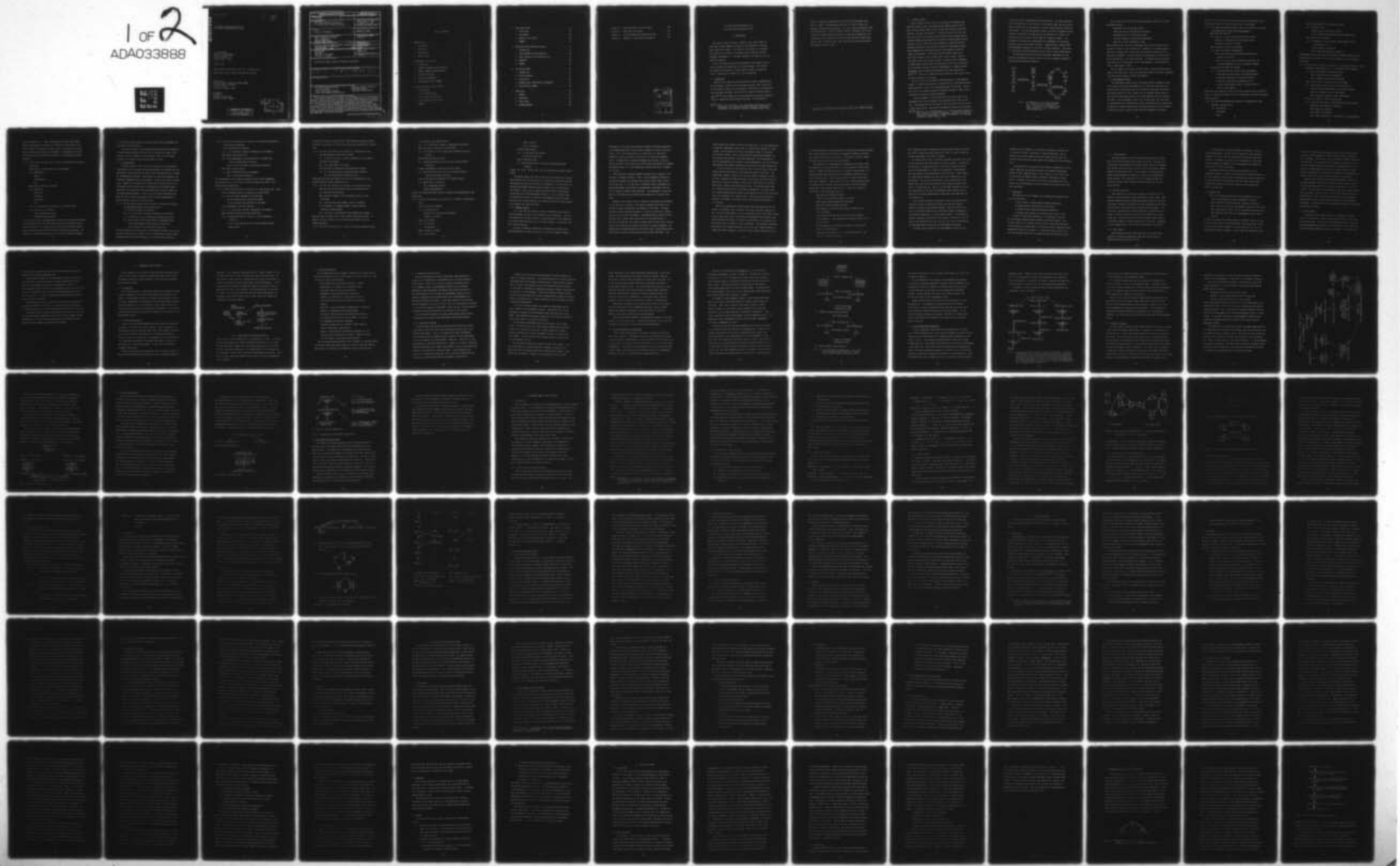
AD-A033 888 WISCONSIN UNIV MADISON DEPT OF COMPUTER SCIENCES F/G 9/2
THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA PROCESSING S--ETC(U)
OCT 76 D R FITZWATER DASG60-76-C-0080
UNCLASSIFIED CSTR-279 NL

1 OF 2
ADA033888



Report CSTR 279

2

# THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA-PROCESSING SYSTEMS

D. R. Fitzwater
University of Wisconsin
Computer Sciences Dept.
1210 W. Dayton St.
Madison, Wisc. 53706

October 1976

Special Report for Period 1 July 1976 - 30 October 1976

Approved for public release; distribution unlimited.

Prepared for
BALLISTIC MISSILE DEFENSE SYSTEMS COMMAND
Contracts Office, BMDSC-CRS
P. O. Box 1500
Huntsville, Alabama 35807

DIR ONR BRD
Room 286
536 South Clark Street
Chicago, Illinois 60605

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. ~~REPORT NUMBER~~
14 CSTR-279

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. ~~TITLE (and Subtitle)~~
6 The Formal Design and Analysis of Distributed Data Processing Systems.

5. TYPE OF REPORT & PERIOD COVERED
Special July 1, 1976 to October 30, 1976

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
10 Donald R. Fitzwater

8. CONTRACT OR GRANT NUMBER(s)
15 DASG60-76-C-0080 new

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Dept. of Computer Sciences
1210 W. Dayton St.
Univ. of Wisconsin
Madison, Wisc. 53706

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
6.33.04.A

11. CONTROLLING OFFICE NAME AND ADDRESS
Ballistic Missile Defense Systems Command
Contracts Office, BMDSC-CRS
P. O. Box 1500
Huntsville, Al 35807

12. ~~REPORT DATE~~
11 30 October 30, 1976

13. ~~NUMBER OF PAGES~~
12 172 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
DIR ONR BRD
Room 286
536 South Clark Street
Chicago, Il. 60605

15. ~~SECURITY~~ CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

9 Special rept. 1 Jul-30 Oct 76,

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed data processing
Real-time systems
Formal system specifications
Formal requirements specifications
Analysis of design processes
Real-time testing
Functional specifications

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The research proposal is to support the development of the "science" behind software engineering in order to ensure required system properties, to compare current software engineering techniques, to develop specification for new design and analysis tools, and to demonstrate the practicality of the "science".

A hierarchical design schema will be developed within which formal representations and analyses can be defined and the required solutions can be found. Since "worst case" problems are generally impossible to solve, sufficient design laws or constraints will be developed to ensure solvability of the critical problems.

404 114

## TABLE OF CONTENTS

# The Formal Design and Analysis of
# Distributed Data Processing Systems

## 1. INTRODUCTION

This special report presents a summary of the current status of work under Contract DASG60-76-C-0080 with the Department of Defense, Army, Huntsville, Alabama. The intent of this report is to document the results of the review of the current state-of-the-art critical problems, development of a "top down" approach , and proposals for more detailed solutions.

All of the results here must be considered as preliminary and subject to change and elaboration as this work proceeds. The problems are complex and certainty of results comes (if then) only when the study is completed with respect to a set of properties.

### 1.1 Introduction

Experience has shown that the specification, design, implementation, and development of complex real-time weapons systems, such as ballistic missile defense systems, are very expensive, difficult to test adequately, slow to develop and deploy, and difficult to adapt to changing requirements.[Da 76] The introduction of distributed data processing concepts potentiallv complicates these problems even more. No development of

[Da 76] Davis, C. G., and Vick, C. R. "The Software Development System," Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California (October 1976), p. 60.

science or design and testing tools will make such development automatic or simple. The engineering decisions will remain complex and dependent on experience and analysis. As in all other engineering fields, the development of a science of design, however incomplete, can be very valuable in guiding engineering decisions, analyzing consequences, providing design laws sufficient to guarantee some desirable system invariancies, and in avoiding "worst case" type designs thus making possible more powerful analysis tools.

Sponsored by The United States Army under Contract No. DASG60-76-C-0080

### 1.1.1 Critical Issues

There are many critical issues in the process of developing and deploying a major system. We cannot hope to address directly most of them. We must address some of them in a way that minimally constrains potential solutions to the others. If we can't solve all of the problems (and we can't), we mustn't prevent others from doing their best on unsolved problems while expoiting our results on those problems we have attacked. We will look first at the problems of making and testing engineering decisions with the goal of making the development process more manageable and engineering decisions more testable at earlier stages of development. We believe that such results could have a major (but indirect) impact on many other critical issues. Even if this were not the case, we believe that a relatively minor investment in better models, design laws, and testing tools will have a large payoff in both the resources involved in development, in speeding up the deployment, and in improving the adaptability of the resulting system[Dr 76]. The final validation of this belief will rest on use of the processes developed in this work.

We will first give an informal characterization of a developmental process. A formal specification of a particular set of desirable developmental processes is one of the intermediate goals of this work. We will start with a very general concept and gradually develop it into a formal and practical scheme for system development.

If the development is to be formalized, the current state of the development must be well defined at the beginning and at least at the end. The passage from one well-defined state to the next will be called a step of the process. Most useful developmental processes will proceed

[Dr 76] Dreyfus, J. M., and Karacsony, P. J. "The Preliminary Design as a Key to Successful Software Development," Proceedings of the 2nd International Conference of Software Development," San Francisco, California (October 1976), p. 206.

via many successive intermediate well-defined states. Of course there may exist any algorithm for carrying out a given process step, and most developmental processes are not guaranteed to succeed, given arbitrary originating requirements. Thus the developmental process is similar to ordinary digital processes, except that its steps may not be effective (i.e. there is no automatic way to carry out a given step) as is shown in Figure 1. Developmental processes may be decomposed into independent or interacting processes, just as can digital processes. Indeed the most general model of developmental processes is just that of digital processes. The notion of "interaction" must be formalized as well as that of well-defined state.

Of course many digital processes do not model a desirable (or even feasible) developmental process; we must still sort through our models to find those with desirable properties.

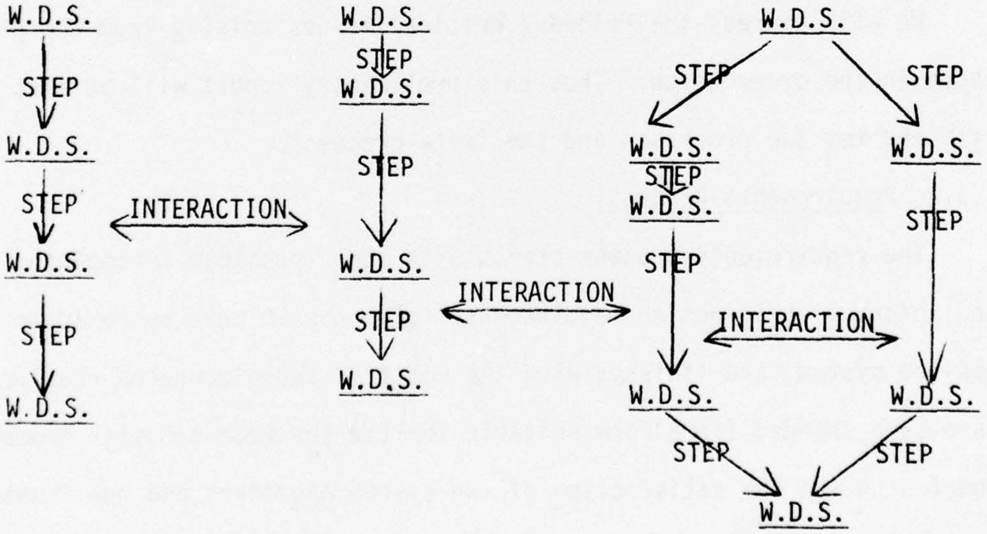


FIG. 1: An interacting set of digital processes with interacting and non-interacting steps. The process is effective if each step is effective. W.D.S. represents "well-defined state".

We can identify at least the following processes involved in a system development process:

. Requirement (of the designed system)

. Design (of the specification of the system)

. Implementation (of the specified system)

. Evolution (changes in design during operation)

. Operation (of the implemented system).

Each process has its own unique requirements for kinds of engineering decisions, of analysis, and of testability. However, all of these processes have much in common, and a general model with properties useful to all of them can be developed. At that high level of abstraction, all such results can be used for any of the above processes. In developing such an abstraction, we must find requirements for all such processes. Finding these becomes a "meta-process" itself.

We will address the relevant critical issues arising from the processes above in the order given. Thus this preliminary report will be most concerned with the top two processes and the "meta-process".

### 1.1.2 Requirements Process

The requirements process starts with some (possibly incomplete, vague, and informal) originating requirements for a system that approximate the desired system, and finishes when the modified and elaborated requirements have been encoded (in a form suitable for the subsequent design process) and tested (to the satisfaction of the system engineers and the "customer"). The steps in the requirements process are of three types: the elaboration of requirements for an approximating system, the modification of requirements to those for a better approximation of the desired system, and the decomposition of the overall desired system into more manageable sub-systems.

There are critical issues involved with both the starting point and the ending point as well as for each type of step in the process.

1.1.2.1 At least the following critical issues are involved in the start- and ending points of the requirements process:

- What should be required?

  e.g., Behavior in real world being affected by system,<br>
  Behavior at interface of real world and system<br>
  Functional structure of system

- What attributes should be constrained?

  e.g., Which are bound by requirements?<br>
  Which are strongly coupled (engineering decisions cannot be factored)?<br>
  Which are loosely coupled (engineering decisions can be made independently within a parametric range)?

- How should requirements be encoded?

  e.g., To make possible the testing of the requirements<br>
  To make possible the design of the required system<br>
  To maximize applicability of tools and analysis<br>
  To check for consistency, completeness, etc.<br>
  To ensure that designed system is testable with respect to requirements.

Under worst case conditions, *none of the above questions have satisfactory* answers, so we must find design laws for requirements that make satisfactory answers possible.

1.1.2.2 At least the following are involved in a decomposition step:

- Decompose into what parts?

  e.g., Abstractions<br>
  Sub-systems<br>
  Levels

. How can requirements be allocated and tested?

e.g., Testable in part

Testable only on collection of parts

Coupling between engineering decisions between parts

. How can parts be integrated?

e.g., At completion of which process (requirements, design, implementation, etc.)

At which stage of elaboration

With minimal testing during integration

The decomposition must consider the subsequent integration and must make integration possible while maximizing the probability that requirements will be met.

1.1.2.3 At least the following are involved in a "better approximation" step:

. How good an approximation is it? (It should satisfy customer.)

e.g., How to test behavior of required system

How to compare with desired behaviors

How to compare two approximating systems

. How can deficiences of approximation be corrected?

e.g., Given analysis data, what are the deficiencies?

Should we change requirements or the desired system?

What can we change to correct deficiency?

What is impact of change on other requirements?

. How can the impact of change be determined?

e.g., Either desired system or approximating system may change

What other requirements are impacted?

Which need to be changed?

Which need to be retested?

What is the traceability of requirements and interactions?

Since the question of "...better approximation to what?" must remain formally unanswered (or we would just change our starting point to an earlier form of requirement), we can not hope to use formal correctness proof techniques on originating requirements. Consequently we can only maximize the analysis information available to the customer and system engineers.

1.1.2.4 At least the following are involved in an elaboration of requirements step:

. Which process is constrained by the requirement?

e.g., Requirements<br>
Design<br>
Implementation<br>
Etc.

. What type of testing is required?

e.g., Stochastic<br>
Analytical<br>
Simulation<br>
Operational<br>
Etc.

. How can requirements be elaborated to a testable level?

e.g., Which parts to elaborate<br>
How to elaborate only part<br>
Minimal design exploration

It is clear that, for a system engineer, the nature of this type of step is essentially the same as for later design steps on the resulting requirement specification. Thus there can be no qualitative dividing line between requirements specification and design specification processes. If possible, the requirements process must be carried far enough in design to satisfy the system engineers (and the customer) that requirements are satisfactory and can be met.

The issues involved in requirement specifications are here addressed in quite general and imprecise terms just to indicate the scope of the problems. We will elaborate on many of these issues in a more formal way after we introduce the appropriate models and tools.

### 1.1.3 Design Process

The design process "starts" with some encoding of the requirements (satisfactory to the customer and systems engineer) and completes with the production of some encoding which meets design requirements and is suitable for implementation designers. Requirements may have been decomposed into relatively independent design requirements. There are six types of design steps: change of requirements, the elaboration of design decisions, the optimization of the specification, the decomposition into more manageable parts, the integration of the decomposed parts, and the interaction with other design processes. There are critical issues associated with both the initial design requirements and the resulting implementation specifications, as well as the design steps.

1.1.3.1 At least the following critical issues are involved in the starting and ending points of the design process:

- Is the state of design well defined?

  e.g., Is it consistent, complete, unambiguous and testable?

- Is it suitable as input to subsequent development processes?

  e.g., Can we decode information needed for development step?

  Can the development process steps be carried out?

  Can we decide if they have been carried out?

We can not hope to give algorithms for developmental processes, but we can at least insist that the current state of a developmental process is well defined and that we can decide when a given step has been carried out.

1.1.3.2 At least the following are involved in the interaction between decomposed design processes:

. Is the interaction well defined?

e.g., Is it consistent, complete, unambiguous, and testable?

. When should interaction occur?

e.g., What independence of design decisions is allowed with loosely coupled design processes?

How to test for and when to synchronize interaction

. What should interaction be?

e.g., What information should be exchanged?

When is interaction completed?

The decomposition of the system requirements leads to semi-independent design processes, but some loosely coupled attributes may require inter-design process interactions.

1.1.3.3 At least the following are involved in an integration process step:

. How can decomposed system be integrated into one system?

e.g., Are both simply connected by interactions?

Does one interpretively simulate the other?

Is one translated into processes of the other?

. Is the integrated system well defined?

e.g. Is it consistent, complete, unambiguous, and testable?

. Does integration preserve designed properties?

e.g. If separate systems are "correct", is the integration "correct"?

Is it at least probable that individual design decisions remain valid?

Since we will control, as part of our formal methods, the types of decompositions, we can hope to resolve these issues with algorithms for integration.

1.1.3.4 At least the following are involved in a decomposition step:

. Are the decomposed systems well defined?

e.g. Are they consistent, complete, unambiguous, and testable?

. Can they be integrated?

e.g. Can issues above be resolved?

. Can associated non-decomposable requirements be tested?

e.g. For consistency during decomposed design processes

For satisfaction after integration step.

There may exist some requirements that can not be decomposed and can only be tested at system integration time.

1.1.3.5 At least the following are involved in an optimization step:

. What are invariancies characterizing an equivalence class?

e.g. What must be preserved?

. What property of members of the equivalence class is to be optimized?

e.g. Given two equivalent members, which is prefered?

. Is it decidable if a proposed member is better than the currently designed member?

e.g. Can we quit an optimization step because we are ahead?

We may be able to require more of some specific optimization steps. In general, we can not require less and know when the steps can be considered completed.

1.1.3.6 At least the following are involved in a design elaboration step:

- Is the result a well-defined sytem?

  e.g. Is it consistent, complete, unambiguous and testable?

- Does it preserve validity of previous tests?

  e.g. Does validity of this system imply validity of previous system?

- What design decisions are made?

  e.g. Elaboration encodes design decision in more detailed structures.

- To what requirements are design decisions traced?

  e.g. What tests must be carried out on result of step to verify and validate decisions?

- How can the design by elaborated to a testable level?

  e.g. Which parts to elaborate
  How to elaborate only part
  Minimal implementation

Ideally, all design decisions should be testable at the completion of the design step.

1.1.3.7 At least the following are involved in a "change in requirements" step:

- What originates the change?

  e.g. Design decisions
  Changes in an originating requirement
  Correction of error

- What is the change?

  e.g. How defined?
  How testable?

- What is impact of change?

  e.g. Local to step

Local to design

Local to subsystem

. Should change be made?

e.g. Is cure worse than disease?

Are there alternatives?

. How can change be made ?

e.g. With minimal impact on current and completed design processes

Changes will occur. We must deal with them effectively and with minimal impact.

The design process ends when the designer has produced well defined specifications for the systems that will collectively meet design process requirements and that can plausibly be implemented. This may require exploratory development beyond the design process prior to producing the final design. Thus we can not hope to draw neat formal lines between the various parts of the development processes. We can, however, hope to characterize the steps of the entire developmental process, leaving decisions such as where dividing lines should be drawn for a given project to the manager who should make them.

### 1.1.4 Computer Sciences

It is clear that the computer sciences are potentially as useful to system developers as physics and mathematics are to engineers. Little of this potentiality has been realized in practice because of the complexity of current systems and because of some unique problems in justifying scientific systems research.

The areas in computer sciences have developed only recently from application efforts in many disciplines. As a result computer sciences departments in each university have been created in differing administrative frameworks and with a wide variety of emphasis on areas of specialization. This is not surprising for a science that is so new and so potentially useful to society, as well as to the university community itself. Although there is general agreement that the area, as an academic discipline, exists quite apart from the many applications of computers to problem solving, it is only beginning to justify the use of "sciences" in its title.

There seems to be general agreement that the area of "systems" is at the heart of computer sciences. There is little agreement on what constitutes the systems area and what the university effort should contribute to this area. It is clear that a substantial body of knowledge concerning the design, specification, implementation, measurement, and control of digital systems has been developed. Such knowledge is potentially useful in providing tools and application systems for all users of computers.

Because of the current lack of a consensus on the nature and standards of this new area of "systems", some special problems have arisen in exploiting research contributions by professors in this area. The major problem in systems research evaluation is intrinsic to the area. Perhaps an analogy will clarify the problem. A mathematician may study artificial universes with a formal rigor that carries its own justification. A physicist may study our real universe without hope of formal rigor, and may justify his studies by the insights and control of natural phenomena. An engineer may use both mathematical and physical tools in designing applications useful to society, and may justify them by that usefulness. Computer sciences has analogs to each of the above areas. For the mathematician we have the "foundations" area, which is concerned with problems of formal systems. For the physicist we have a developing science of the design of artificial systems, which is concerned with technology-independent system universes too complex to have been formalized. For the engineer we have the implementation of systems in given technologies. We can distinguish both hardware and software engineering as subfields of systems engineering.

The justification of the foundations area is much the same as for mathematics, and the justification of the engineering areas is as usual. The system area, however, suffers from a serious problem in finding its justification. Because the system universes are artificial, we can not say (as does physics) that any insight or control into that universe is justified. Physics has a unique, self-justifying universe. The system research does not. Neither can such work be justified as mathematics, since the artificial system universes being studied are too complex (so far) to be formalized. The system researcher has a double burden. He must not only justify his solution to a problem, he must also justify the universe within which it is a solution.

Most of the contemporary systems research has been carried out in the context of different, local, universes--the locally available computer system. Each such system defines a set of constraints which creates many problems local to that system. This localization has fragmented system research into rather isolated user groups. The problems and solutions of one group are of little direct use or interest to other such groups. The informed, interested peer group to such system researchers may be very small indeed, perhaps including only local co-workers. With time, these isolated efforts will significantly contribute to the contribution of more abstract (machine-independent) system universes within which a substantial community of workers may produce broadly applicable results. A universe is neither created nor justified in a day or by a few applications.

The complexity of requirements for contemporary and future system developments is so great that there is serious question of how to deal with it. Indeed, the principle result of computer sciences today is to demonstrate how impossible worst-case developmental processes are to specify and carry out. That is not much help. We can not deal with worst-case complexity. Thus we must accept as a primary postulate that if something can't be done, then we must accept sufficient effective design laws so that we don't have to do it. There are a number of corollaries to this postulate, such as:

- Never work with arbitrary systems
- Encode so that required decoding is possible
- Never accept unconstrained design decisions
- Generate only well-behaved structures
- Only replace a restrictive design law by another that is still sufficient.
- Satisfaction of design law must be practically testable
- If an algorithm is too computationally complex, then it can't be carried out.
- We must constrain the system being required, not just the form or its description.
- Detection of arbitrary cases is usually not possible. Prevention is then the only cure.

One of the major reasons contemporary system engineers have not resolved the critical issues mentioned previously is that, in terms of arbitrary systems requirements, they have no solution.

We resolve these issues if we accept the above postulate , but at the risk of restricting the system domain to trivial systems. A major goal of this work is to show that we don't really lose anything essential from our systems domain by accepting that postulate. Unfortunately, much work remains to be done before we reach this goal. Surprisingly, our major hope of reaching this goal rests on the complexity of the applications. We can not hope to analyze value manipulations in any general way, and so we must encode most requirements and design decisions in a structural way. Even so, the intrinsic problems are so complex that unless we can make the development steps formally trivial, they will be at least computationally impossible, even algorithms are not enough for us.

Thus our major problems are extrinsic (that of how to define the problem structures and processes) rather than intrinsic (that of finding a solution to the defined problems). We can't possibly use most of the sophisticated analysis techniques because of combinatoric computational complexities in large complex systems. If we define our problems correctly, the solutions are not hard. We may balk a bit at some of the resulting design laws, but this will serve as a motivation for developing better design laws and developmental processes.

In effect, we must approach the developmental process not as a mathmatician (the complexity is too great for analytic solutions) but as a physical scientist interested in system invariancies. We can develop a hierarchical approach that will ensure the most basic and essential design properties and provide the foundation for future elaboration.

The formulation of such design laws provides a common basis for surveying, comparing, and evaluating current and proposed methodologies. Indeed, the most immediate impact of this study will lie in such critical evaluation and identification of potential improvements of high payoff. Ultimately, the system restrictions should make possible the creation of new developmental processes and tools applicable within that restricted domain.

## 1.2 Objectives

The objectives of this proposal are to develop the following hierarchical approaches:

- to the "science" of design behind software engineering,
- to demonstrate the practicality of the science,
- to compare current software engineering approaches,
- to develop specifications for new design and analysis tools.

In order to reach these objectives, it is necessary to develop design schemata, required system properties, and formal models. Each of these areas will, of course, require cycles of study and elaboration into hierarchies of greater depth. No one area could be completed without commensurate studies on the others. A significant amount of work in at least the following areas will be required to meet the objectives above.

### 1.2.1 Design Schemata

The design process itself must be formalized to provide a framework within which problems can be studied and solved. The goal is to allow maximum factorization of the design process itself into independent designs, while providing a suitable level in the design schema for making all required decisions. For each design, a suitable design schema should encompass the functional, process, and implementation design problems. The formalization of such a schema requires the creation of system universe models within which decisions and laws may be defined.

### 1.2.2 Required Properties

Some properties of a particular system are valid only for that system and must be ensured by means appropriate to that system. Many required properties seriously impacting the possibility of achieving performance, integrity, evolution, and design automation can be identified and used to drive the creation of design laws and models. The hierarchical classification of these properties is essential for designers to select the level of the design laws appropriate to each design step. The certification of real time systems alone, requires a substantial number of required properties to be present. Debugging can only display errors, rather than show that there are no errors.

### 1.2.3 Formal Models

Each hierarchical design schema applied to a factored design problem will require an appropriate model and formal system for representing design laws and decisions.

It is clear that the design problems are insoluble in terms of unconstrained models. We must find the reasons for such impossible solutions and "pass" effective design laws sufficient to make them solvable. Ultimately, such laws will be incorporated in high level design and analysis languages to free the designer from unnecessary details, and to ensure consistency.

The models and design laws should be sufficient to allow algorithmic analysis of design to show presence of required properties, to allow creation of useful tools and techniques for design otimization at each level and for translation to the next level, and to reduce computational complexity of design, analysis, and transformation tools to practicality.

## 1.3 RESEARCH PLAN

There are five major problems that must be addressed as a set:

- What are developmental process requirements in general?
- What hierarchy of system properties should be established?
- What sufficient design laws are needed to support the hierarchy?
- What developmental process can best incorporate the design laws?
- What encoding of the state of a developmental process should be used?

To limit the scope of the current work in a "top down" fashion, we will restrict the hierarchy of system properties to be studied here to only a few, most basic ones relevant to critical issues of real-time, distributed data processing systems. We will treat the developmental process by first considering requirements specification and then design processes. After completion of this restricted study of the above set of problems, we will have a basis from which to address the remaining objectives. At that point we will develop a research plan to extend the relevant hierarchy of properties, to extend study to later stages of the developmental processes, to compare potential system engineering approaches, to develop specifications for new design and analysis tools, and to assess potential payoff for the results.

Because of the tight coupling between the potential solutions to the above problems, we can not solve them in order. We will instead develop approximate solutions to each and iterate until all are consistent. This iteration is not completed as yet and so current results must be tentative. We will do this first for the requirement specification process. Even so, we must less formally explore subsequent developmental process stages to assess the validity of requirements methodology. In effect we are in a developmental process for developing developmental processes.

## 1.4 SPECIAL REPORT

The purpose of this special report is to document the current status of the research in order to consolidate and form a firmer basis for the future work, and to facilitate the decomposition of this work into more manageable sub-problems. Up to this point, the concepts and results have been changing too rapidly to document. This report contains an overall framework and enough detail to identify and factor the problem into relatively independent parts.

We will develop a model for the requirements process in section 2, and a model with a detailed instance of a formal specification of the state of a developmental process will be presented in section 3. A brief characterization of real-time data-processing systems from the point of view of requirements specifications and relevant properties will be given in section 4. A similar treatment of distributed data processing systems is given in section 5.

With the previous results in hand, we will then turn our attention to the design process in section 6. Our current conclusions and plans for remaining work on this current contract will be discussed in section 7.

Of course, results from these later sections will have to be fed back into the requirements methodology and used to critically compare current methodologies. This has not been carried out as yet because we first needed the firmer foundation provided by this report.

# 2. REQUIREMENT SPECIFICATIONS

We will develop, in this section, a model for the requirements specification process based on general principles applicable to any system. Subsequent sections will extend the model to real-time and distributed data-processing systems.

## 2.1 Introduction

The starting point for a requirements process must inevitably be some informal, incomplete, and possibly inconsistent desired behavior for a system. The requirements process must produce a well-defined, complete, consistent, unambiguous, and testable set of system requirements ensuring that system behavior will be satisfactory to the customer and that the required system can plausibly be designed and implemented. The resulting requirements specification must also be suitable for the remainder of the developmental process.

### 2.1.1 Formal Approximations

Clearly we can do nothing formally with the originating informal requirements; we face the "worst-case" problems. After development of requirements design laws from our later study of developmental processes we can hope to improve the suitability of originating requirements. After all, there is no unique set of requirements for a given system and if we understand what makes one set better than another, we could use the better set to avoid some problems. In any case, the following arguments remain valid.

Clearly we must develop an approximate set of requirements which we then refine by elaboration and testing until the requirements process is completed. If we interpret "approximate set" as a vague, informal set, we can offer little or no formal assistance in carrying out the process. We can, however, require that the "approximation set" precisely and formally specify a system whose behavior approximates the desired behavior. In this case we can offer substantial formal assistance to the process since the approximating system is well defined. The resulting process is described by Figure 2. We can thus proceed from one well-defined requirement set to another, using formal analysis tools at each step, until the desired requirements have been produced.

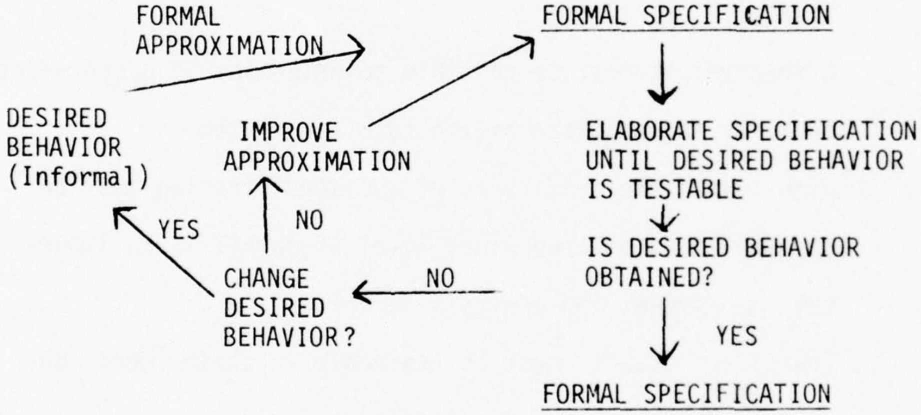


FIG. 2: REQUIREMENTS SPECIFICATIONS PROCESS

We do not hope to deal formally with all attributes of a system. Only some will be constrained by our formal requirements specification. The remainder are not formally treated, and any desired methodology can be utilized for their elaboration and testing. Our formal tools will not replace all other methods. In fact, their role will be supplemental and optional. They may very likely become the framework within which an arbitrary methodology can be embedded.

### 2.1.2 Required Properties

We can immediately derive a number of properties our formal requirements specification must have if this process is to be carried out. Some of these are clearly the following:

- Well-defined: we must be able to decide if proposed approximation is suitable for our analysis tools.
- Consistent: behavior must be well-defined.
- Unambiguous: which behavior must be specified.
- Complete: all behavior, at a given level of detail, must be specified for those attributes to be formally treated.
- Effective: it must be possible to determine if a specified behavior is possessed by the required system.
- High level: the primitives of the specification must be as complex as desired since level of detail is an important management and analysis parameter.
- Traceable: changes must be traceable in their impact on current "approximate" specification.
- General: system engineers should not be unnecessarily constrained in the systems they can require. Some constraints will be necessary to avoid worst-case problems.

We can provide automated analysis tools to decide if a proposed approximation does indeed have these properties. We will accept the need for these tools as a constraint in developing our formal specifications.

### 2.1.3 Elaborating Specifications

We may use combinations of any of the process steps described in section 1.1.2 to elaborate the requirements until the desired behavior can be tested. Clearly this could be a complex and lengthy process. It is, however, simply the developmental process we are producing. Potentially, each approximate requirements specification may have to be carried arbitrarily far through the developmental process prior to testing. We would like to develop design laws and developmental processes that minimize such effort. The primary attack will be to factor and only elaborate to the level of detail specifically required for testing. Details of this approach are developed in later sections, and some forms of decomposition and partitioning will be discussed in this section.

We will be able to exploit our entire formal design process results in assisting in this part of the requirements specification process.

### 2.1.4 Testable Requirements

The requirement that the specification must be effective has significant impact on both the form of the specification and the form of the informal requirements. There is no problem with simple requirements which can be tested in an ad hoc analytic fashion, but most requirements will be too complex to be tested without simulation. "Effective" is thus equivalent to a simulation specification. Ideally, all types of simulation should be applicable to our formal specification, and our requirement specification is equivalent to a simulation model. This equivalence must either be an identity or there must exist a practical algorithm for deriving the simulation model from the requirements specification.

Because of our data processing background, we view the simulation model as a digital simulation. The extreme versatility of digital simulation techniques made this hypothesis plausible even if some suspicion remained that it was not a satisfactory decision for non-data-processing systems. A closer look at contemporary requirements processes, however, substantiates the validity of the hypothesis. Even radar designers frequently prefer to start with (and elaborate) a digital simulation that encodes the requirements in an effective way. Digital simulation is the most common tool used in developing requirements.

It is also worth noting that the testability requirement applies to all stages of the developmental process. Thus we have not, as yet, developed any reason for using different formal specifications at each stage. The difference from stage to stage lies in the level of specified detail rather than in a qualitative difference.

The originating requirements may be for an open or for a closed system. The specification of a closed system includes the "environment" in which all interactions take place. An open system may include an interface specification to the "environment" but does not include that "environment". An open system is not testable without a specification of an environmental driver.

The requirements specification must be closed to be testable. This may be done either by specifying system, interface, and driver or by simply specifying the closed system that includes the environment. Both options must be possible. However, we can maximize the scope of our formal techniques if the formal requirements approximation is the closed system. The partitioning of the closed system into system, interface, and "driver" could be done subsequently in a formal way if desired. Such a partitioning may not always be feasible or desirable, particularly at the initial informal level. The closed system specification is a complete and consistent specification whereas the open system, interface, and driver specification (informally arrived at) need be neither complete nor consistent and undetectably so in the worst case. We will, therefore, start our formal specification as that of a closed system.

Requirements elaboration can thus take place jointly in the environment and system models with a homogeneous technique. We can "design" and test the detailed environment at the same time and in the same manner as the system interacting with the environment.

Thus we can provide substantial and automated assistance in verifying behavior of the required system. Indeed the requirements process does not end before the customer is satisfied with displayed behavior.

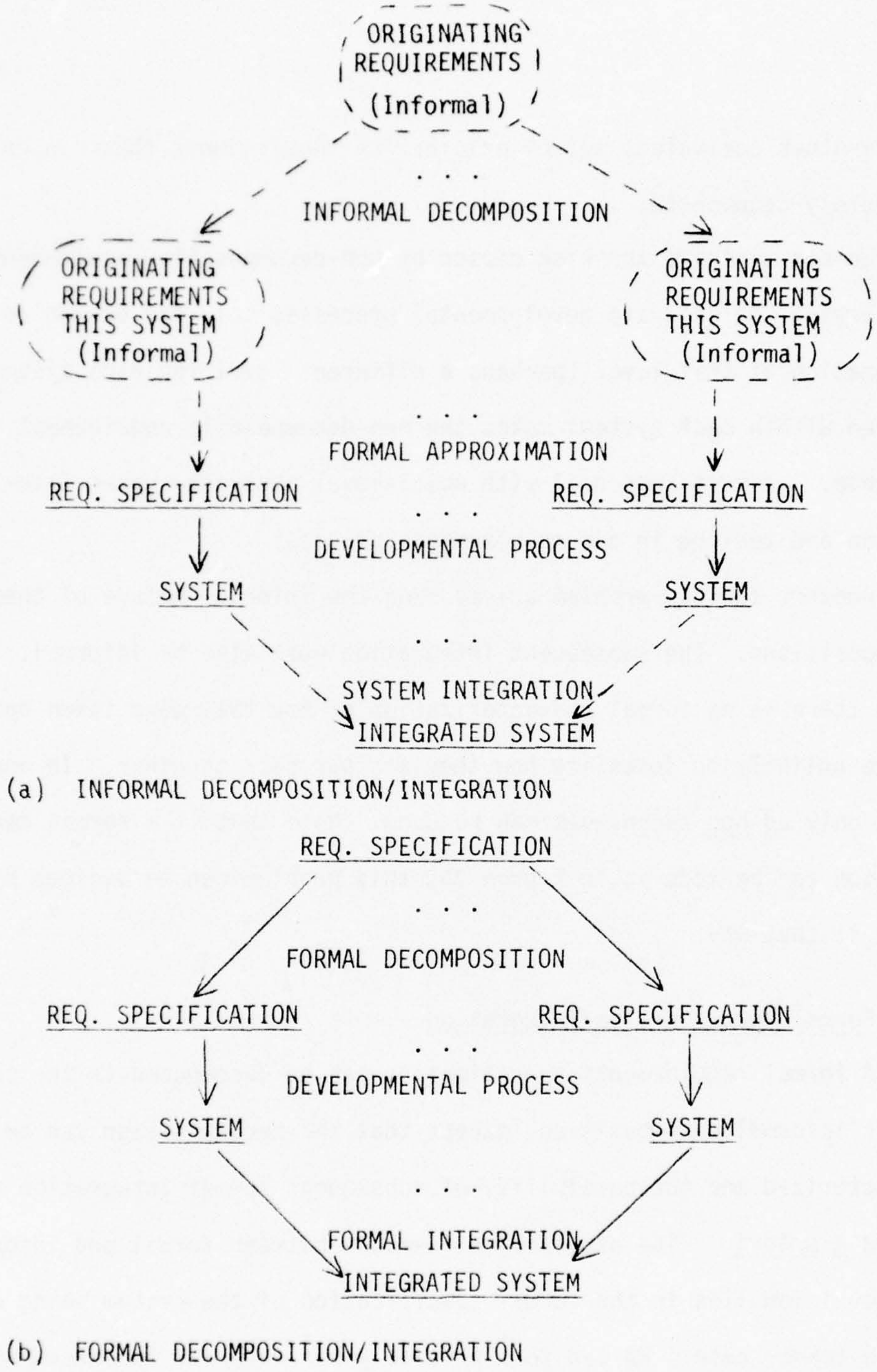
## 2.2 Informal Decomposition/Integration

A large system development will have many attributes that are only very loosely coupled, for example, the physical site design and the data processing design. Design decisions local to different design processes are not sensitively coupled and, within predesignated limits, may be made quite independently. Thus some forms of decomposition can be carried out at the earliest stages, thus factoring the development process into independent processes until the corresponding system integration occurs.

The earliest possible point of decomposition is in the informal originating requirements as shown in Figure 3a. Following the arguments of section 2.1.4, we will decompose into closed systems by identifying tightly coupled subsets of attributes that are only loosely coupled with other subsets. Then by partitioning the attributes we can specify a closed system approximation for each attribute subset. The decomposition of a closed system is thus into a set of closed systems, each representing the original system from a different viewpoint.

The number of such decomposed systems is clearly application-dependent and sensitive to the identification of loosely coupled subsets of attributes. Some, such as physical site systems, logistic systems, logical systems, etc. are clearly loosely coupled since, within predetermined limits, design decisions are essentially independent. Those limits can be built into the requirements for each decomposed system. The application of our developmental process may now be made independently (and tested independently as well).

There may be some originating requirements that can not be analyzed for attribute coupling that prevent such decompositions. Since the originating requirements are not usually unique expressions, perhaps one can find a better set of requirements which can be decomposed. In any case, some non-decomposable requirements may still remain. These requirements thus can only be tested on the resulting integrated system. If such a requirement is tightly coupled to attributes scattered across the decomposed system, then the decomposition itself is probably ill-advised since system integration may produce expensive test failures. Perhaps one can



(a) INFORMAL DECOMPOSITION/INTEGRATION

(b) FORMAL DECOMPOSITION/INTEGRATION

FIG. 3: System decomposition/integration. Solid lines represent formally defined entities. Dashed lines represent informally defined entities.

find another equivalent set of originating requirements that can be more completely decomposed.

We can minimize the risk caused by non-decomposable requirements by carrying the separate developmental processes only far enough so that integration at that level (perhaps a different level for each system, or even within each system) makes the non-decomposable requirement testable. We must then deal with multi-level abstract system integration and testing in our requirements process.

Another serious problem arises from the informal nature of these decompositions. The subsequent integration must also be informal. Since there is no formal characterization of how they were taken apart, we are unlikely to formalize how they are put back together. In any event only ad hoc techniques can be used. Note that if a formal decomposition can be made as in Figure 3b, this problem can be avoided by doing it that way.

## 2.3 Formal Decomposition/Integration

A formal requirements specification can be decomposed in the same way as informal decomposition, except that the decomposition can be formally characterized and the possibility of subsequent formal integration can be tested a priori. The essential difference between formal and informal decomposition lies in the formal specification of the system being decomposed in the former case. We can thus precisely characterize the decomposition (even if we have no effective procedure for carrying it out) and establish sufficient conditions to ensure the correctness of the decomposition. Both the decomposition and the integration may be substantially aided by design automation tools. Indeed, we can accept sufficient design laws on the form of the decomposition to ensure that integration can be done and tested automatically. We should still support integration at many levels of design detail to minimize risk from non-decomposable requirements.

An example of a possible data processing system decomposition and integration is given in Figure 4.

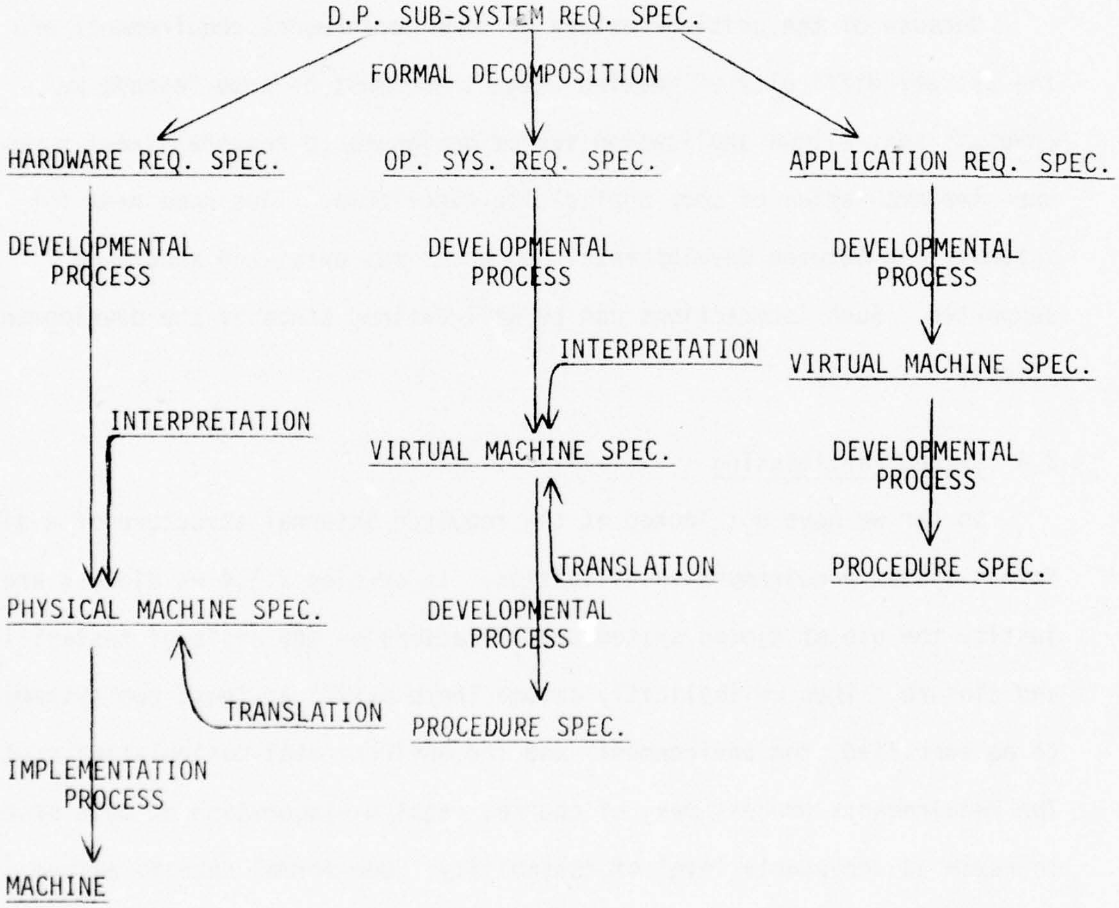


FIG. 4: An example of formal system decomposition/integration. The effective property of each formal specification ensures that the specified system is in a form interpretable by a formally universal interpreter (the simulating system). Translation (compilation) steps may improve efficiency of the testing of resulting implementation. The example developmental process does not exhaust all possibilities but is intended as an illustration.

In this case we can develop translators and translator writing systems to aid the integration of the decomposed systems.

This type of decomposition may be very powerful in isolating the effects of changes to one of the decomposed systems. Similarly, because of the loose coupling between such systems (or they would not be decomposed), some of the components may be directly usable in other applications, or easily adaptable to changes in an application.

Because of the critical nature of some performance requirements and the extreme difficulty of meeting them, there must be some "escape mechanism" that allows application system designers to require direct hardware implementation of some application algorithms. Thus some need for interactions between developmental processes may exist and should be supported. Such interactions can be well-defined steps of the developmental process.

## 2.4 System Partitioning

So far we have not looked at the required internal structure of a single, formal system requirement specification. In section 2.1.4 we discuss and justify the use of closed system specifications on the basis of testability and closure. Thus we implicitly assume there exist at least two systems to be specified, the environment and the environmental-manipulating system. The requirements process may, of course, require elaboration of both systems to reach an acceptable level of testability. Our formal specifications of a system must be able to require an interacting system complex with formal specification, not only of the systems in the complex, but also of their interactions. The partitioning of logical functions among the systems in the complex does not necessarily imply the same partitioning of physical systems in the implementation. This point is illustrated in Figure 5.

The logical partitioning of Figure 5 involves only a part of the formal requirements. For our purposes, we can factor system requirements into the following:

- . Performance: how well must the system work?
- . Resources: what kind and how many can the system use?
- . Logical: what functions must the system support?

We are disregarding here the valid need for similar requirements on the requirements process itself, and we are not formalizing the testing of those requirements. This is a potentially important exception, and the need for better management tools for the developmental process is recognizable. We feel that formalizing the developmental process and the testing of developing systems are essential first steps in solving management problems.

The testability of a partitioned logical requirement specification is still dependent on the existence of the other logical specifications since only then do we have a testable closure. We can use for this purpose the currently most suitable of the other specifications in the partitioned set. Figure 5 thus becomes an elaboration of Figure 2. If the performance and resource requirements are loosely coupled, we could use formal decomposition instead, and make the testing even more factored and localized as in Figure 3, but that would not be a partitioning step.

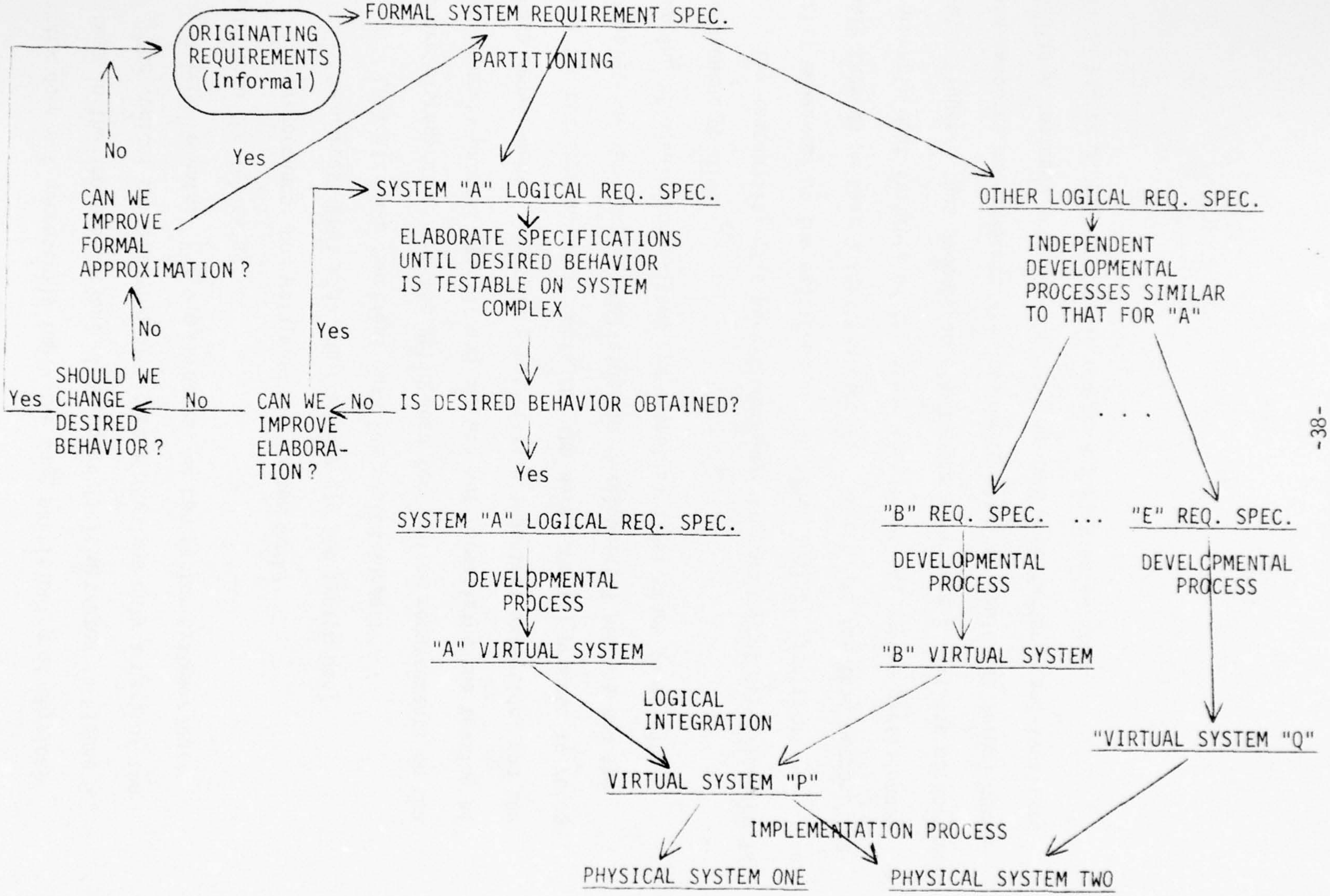

ORIGINATING REQUIREMENTS (Informal)
FORMAL SYSTEM REQUIREMENT SPEC.
PARTITIONING
No
Yes
CAN WE IMPROVE FORMAL APPROXIMATION ?
SYSTEM "A" LOGICAL REQ. SPEC.
ELABORATE SPECIFICATIONS UNTIL DESIRED BEHAVIOR IS TESTABLE ON SYSTEM COMPLEX
OTHER LOGICAL REQ. SPEC.
INDEPENDENT DEVELOPMENTAL PROCESSES SIMILAR TO THAT FOR "A"
No
Yes
SHOULD WE CHANGE DESIRED BEHAVIOR ?
Yes
No
CAN WE IMPROVE ELABORA-TION ?
No
IS DESIRED BEHAVIOR OBTAINED?
Yes
. . .
SYSTEM "A" LOGICAL REQ. SPEC.
"B" REQ. SPEC.
...
"E" REQ. SPEC.
DEVELOPMENTAL PROCESS
DEVELOPMENTAL PROCESS
DEVELOPMENTAL PROCESS
"A" VIRTUAL SYSTEM
"B" VIRTUAL SYSTEM
LOGICAL INTEGRATION
"VIRTUAL SYSTEM "Q"
VIRTUAL SYSTEM "P"
IMPLEMENTATION PROCESS
PHYSICAL SYSTEM ONE
PHYSICAL SYSTEM TWO

FIG. 5: Logical requirements partitioning

We must thus assume that performance and resource requirements are not practically decomposable into relatively independent requirements for each system in the partitioned set. Any attempted decomposition would run into "the requirements allocation problem" which in these terms is insoluble. Thus we won't try to solve it, but we don't need to in order to carry out the requirements process of Figure 5.

We may be able to do even better if we can develop parametric logical specifications for each member of the partitioned set. We can then use the decomposition and integration steps of Figure 6. All testing can now be done independently within parametric ranges.

Each of the decomposed systems is a system complex representing the entire system and is thus a testable entity. Interactions between the decomposed requirements processes are now required only when parameter ranges must be exceeded to meet specifications, and when the partitioned systems are integrated by collecting them into a system complex.

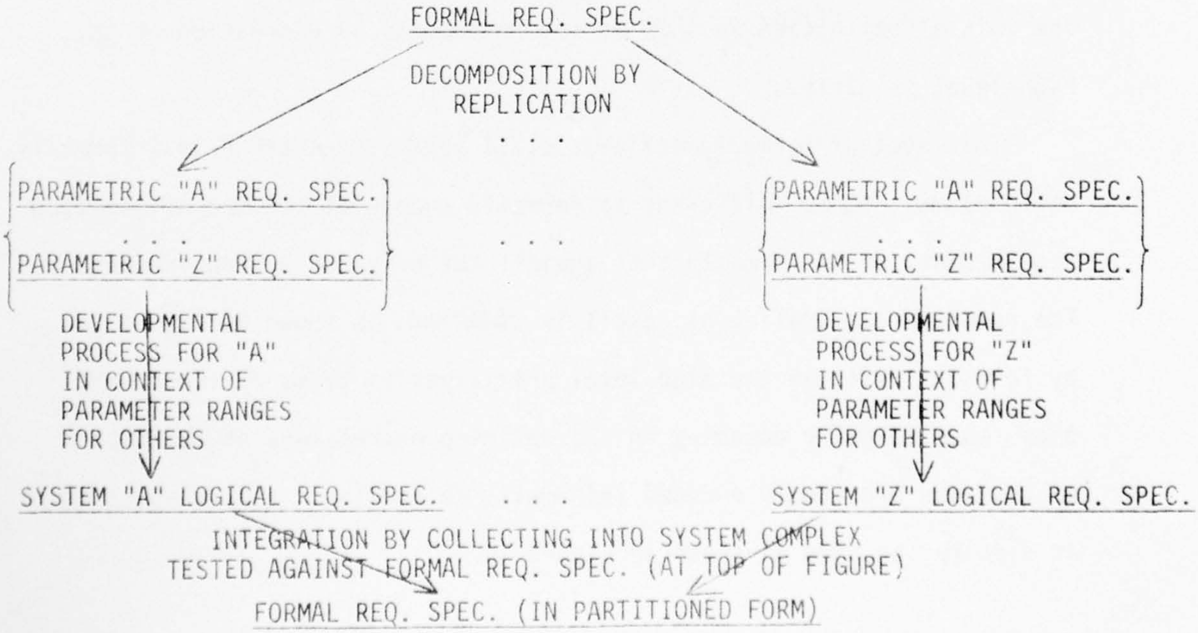

FORMAL REQ. SPEC.
DECOMPOSITION BY REPLICATION
. . .
PARAMETRIC "A" REQ. SPEC.
. . .
PARAMETRIC "Z" REQ. SPEC.
. . .
PARAMETRIC "A" REQ. SPEC.
. . .
PARAMETRIC "Z" REQ. SPEC.
DEVELOPMENTAL PROCESS FOR "A" IN CONTEXT OF PARAMETER RANGES FOR OTHERS
DEVELOPMENTAL PROCESS FOR "Z" IN CONTEXT OF PARAMETER RANGES FOR OTHERS
SYSTEM "A" LOGICAL REQ. SPEC.
. . .
SYSTEM "Z" LOGICAL REQ. SPEC.
INTEGRATION BY COLLECTING INTO SYSTEM COMPLEX
TESTED AGAINST FORMAL REQ. SPEC. (AT TOP OF FIGURE)
FORMAL REQ. SPEC. (IN PARTITIONED FORM)

FIG. 6: Parametric partitioning

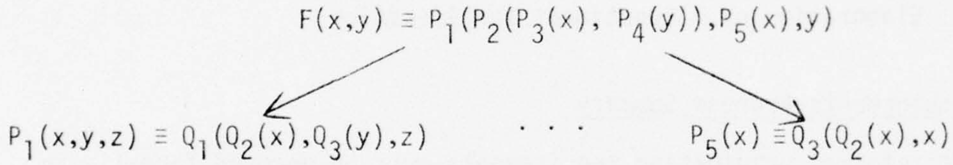
## 2.5 Primitive Elaboration

We will assume that the formal requirements specification will be defined using a functional formalism designed to provide a sufficiently general model for all systems of interest. There previously was no such model available, primarily because of the need to functionally model asynchronous interactions which was not met in available models. We have developed such a model and it is described in section 3. In this section we will ignore the requirements for formal interactions, although after section 3 they can be dealt with without change in the discussion presented here.

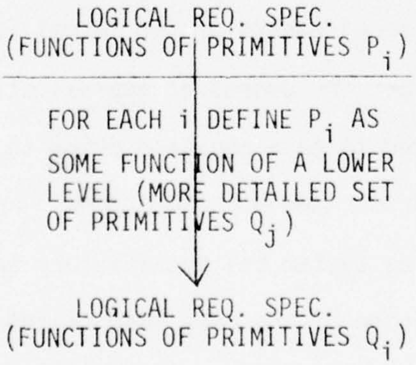
The initial formal requirements specifications will use rather high-level primitive functions in precisely specifying the system. Primitive functions are not formally defined but rather are informally characterized (e.g., in English descriptions). Each such initial primitive may eventually be elaborated by the developmental process into many processes or procedures spread over an entire network of implemented (physical) systems. The initial definition is thus as some mathematical expression of the high-level primitives.

The level of formally defined detail (primitives are only informally defined) may not be sufficient to formally encode all of the originating requirements or to formally test against the originating requirements. The required elaboration of detail is obtained, as shown by Figure 7a, by formally defining the high-level primitives in terms of lower level ones, thus formally encoding in the defining expressions at least part of what was previously encoded informally in English. This results in an elaboration step as shown in Figure 7b.

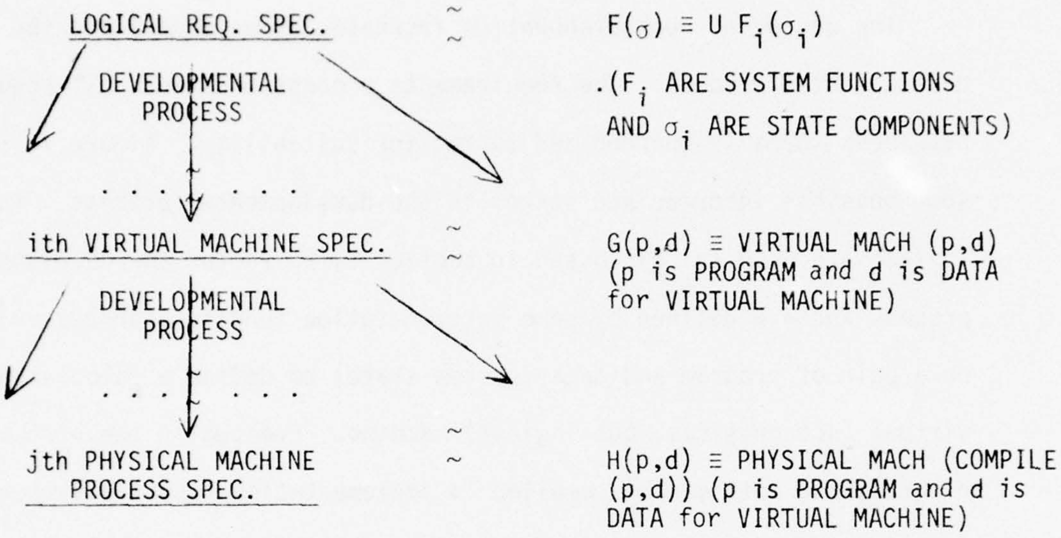
The degree of such elaboration increases as we move along the developmental process. The requirements process ends when all requirements have been formally encoded and tested for suitability. Figure 7c describes some possible intermediate states in the developmental process. The virtu- systems are used (as discussed in section 5) to factor the developmental process and are defined by some interpretation function (processor) operating on a pair of program and data (system state) to define a computation of the virtual (not physical, but logical) machine. Eventually the virtual machine programs and data may be compiled to implementation (physical) machine initializations. This design process is further described in section 6.


$F(x,y) \equiv P_1(P_2(P_3(x), P_4(y)),P_5(x),y)$
$P_1(x,y,z) \equiv Q_1(Q_2(x),Q_3(y),z)$
. . .
$P_5(x) \equiv Q_3(Q_2(x),x)$

(a) Function Definition Tree



(b) Primitive Elaboration Step


LOGICAL REQ. SPEC.
DEVELOPMENTAL PROCESS
. . . . . . .
ith VIRTUAL MACHINE SPEC.
DEVELOPMENTAL PROCESS
. . . . . . .
jth PHYSICAL MACHINE PROCESS SPEC.
~
~
~
$F(\sigma) \equiv \cup F_i(\sigma_i)$
.($F_i$ ARE SYSTEM FUNCTIONS AND $\sigma_i$ ARE STATE COMPONENTS)
$G(p,d) \equiv$ VIRTUAL MACH (p,d) (p is PROGRAM and d is DATA for VIRTUAL MACHINE)
$H(p,d) \equiv$ PHYSICAL MACH (COMPILE (p,d)) (p is PROGRAM and d is DATA for VIRTUAL MACHINE)

(c) Levels of Primitive Elaboration

FIG. 7: Elaboration of a Functional Specification

## 2.6 Requirements Process Summary

The informal originating requirements must be encoded formally in some functional specification whose behavior approximates that of the desired system. The behavior must be testable and, if unsatisfactory, either the originating requirements or the formal functional specification must be changed to improve the degree of approximation. The level of detail formally encoded may need to be elaborated prior to testing. The requirements process ends when the current originating requirements are formally encoded and the specified system has satisfactory behavior. This may have required partial completion of the remainder of the developmental process.

We have identified several types of requirements process steps (e.g., approximation, decomposition, integration, partitioning and elaboration) and discussed the issues involved in their formalization.

We have studied the issues of formal testability and have described an approach to their resolution by the formal, functional, effective specification of the requirements for closed systems.

We have identified a number of important properties a formal system specification must have in general, and laid a foundation for the subsequent work in section 3. Other required properties will be developed, after the formalism is established, as design laws which ensure that the tools and tests discussed in this section can actually be provided. Further design laws will be derived from studies described by the remaining sections. This is the first pass through this material, and feedback of results obtained in later sections will be incorporated in the final report due in January 1977.

# 3. FUNCTIONAL PROCESS SPECIFICATIONS

## 3.1 Introduction

The requirements for a system require needed behaviors or restrict behavior to certain limits. It is the behavior of the system being designed which our formal specifications must relate to the requirements. We must be able to verify that a specification corresponds to required behavior. For this verification we must be able to observe the specification's behavior, which entails the ability to observe the well-defined states and interactions of the specified systems. We may think of these well-defined actions as state transitions of a digital process, and the well-defined interactions as interface transitions. The state transitions can be defined as algorithms (possibly nondetermininstic) for the successor state.

Many required systems have behavior which is primarily factored into the behaviors of several components, particularly for geographically distributed processing. However, these components are required to communicate and coordinate behavior via some form of interactions--the states of one component have an effect on the states of the other component. It is also important that these interactions may occur asynchronously, as the indispensibility of interrupts has shown. We must have a formal model for such complex processes.

### 3.1.1 Basis for Functional Specification

From the goals for modelling system behavior and for observing and verifying specified behavior, we have the concepts of state, algorithmic state transition, and interacting component processes of a system. Our formalism begins with these concepts and develops according the goals and properties required for a specification formalism.

There is a generally accepted concensus that a process can be defined by a set $\Sigma$ of process states and a (possibly nondeterministic) successor function $f$ . The application of $f$ to a process state $\sigma$ to produce a process state $\sigma'$ is known as a process step. When we wish to specify a a more complex process in which internally asynchronous or independent transitions occur, there is no longer a consensus, and more work is needed. Ramamoorthy and So [RAM 76] have said that a functional process specification should be "(1) comprehensible, (2) unambiguous, (3) verifiable, and (4) machine processable". These goals are certainly justified by the need for a requirement methodology: (1), (2), and (3) are needed for correctly and consistently formalizing functional requirements, (2) and (3) are needed for correctly developing and implementing specifications, and (4) is needed for accuracy, for the volume of the work, and for simulation testing. These goals may be met by a formalism based on a mathematical notation of functions and sets which has been subject to constraints which guarantee (3) and (4) and which allow sufficient expressive power for (1).

A further goal is to ensure properties which we desire the specified system transitions to have, such as: algorithmic implication (computations terminate without blocking), the ability to test real time systems by nonreal time simulation, and the ability to model any characteristics of a real system by our specifications (completeness from 2.1.2). In general, a specification should be such that each of a set of relevant properties is

[RAM 76] Ramamoorthy, C.V. and So, H.H., "Survey of Principles and Techniques of Software Requirements and Specifications", Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1976.

either guaranteed by the form of the specification, or is efficiently decidable on it. Furthermore we should be able to limit the information in a specification to that necessary to ensure these properties or their efficient decidability. We would also like our specification formalism to allow expansion of this goal as useful new properties are discovered and included in the relevant set (for example properties relating to the evolutionary development of existing systems).

Specifications based on mathematical function notation allow concentrating on relevant areas of a system and hiding the rest within primitive functions; this is the high level property mentioned in section 2.1.2. Such specifications also permit guaranteeing properties or their easy decidability by axiomatic constraints on function combinations, using much of what is already known about function behavior. Simple primitives for interactions may be inserted into the formalism of mathematical functions quite naturally, and these interaction primitives may also be handled by axiomatic constraints. Such a formalism also ensures the consistency and unambiguity of a specification if only minimal care is taken.

### 3.1.2 Properties of a Specification

In justifying the mathematical form of a functional process specification we have made reference to desirable properties for a specification as well as to more general goals in the development of functional specifications. The properties include:

(1) observable and verifiable behavior of a specification.

(2) generality for asynchronously interacting processes.

(3) algorithmic implication (all state transitions will complete).

(4) testability--particularly of real time distributed processes by simulation.

(5) completeness of specification with respect to characteristics of required system.

(6) ability to superimpose developmental and evolutionary processes on the specification formalism.

(7) ability to concentrate only on areas of the system relevant to desired analysis, leaving low-level details within primitives.

(8) consistency and unambiquity of specification.

In addition to these properties there are two mentioned in section 2.1.2:

(9) effective decidability of the behavior of a system specification.

(10) traceability of the impact of changes in a system specification.

The decidability of behavior must be in terms of analyzing asynchronous interactions, which will be discussed later. Tracing the impact of changes depends upon the change being local to an area of the specification, which in turn depends upon a correct design decision in factoring the specification.

## 3.2 System Specifications

We start our discussion of formal specifications by introducing some basic definitions. (A more rigorous and complete treatment is given in Appendix A.)

Definition. A value space $V$ is a set of values $v$ which are not here further defined.

Definition. A state component $\sigma_i$ is a subset of a value space $V_i$ .

Definition. A state component space $\Sigma_i$ is the set of all state components $\sigma_i$ which are subsets of the value space $V_i$ .

Definition. A state space $\Sigma$ is a product $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m$ of state component spaces, i.e. $\sigma \varepsilon \Sigma$ if and only if $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_m)$ where $\sigma_i \varepsilon \Sigma_i$ , $i = 1,2,\cdots,m$ .

Definition. A process is a pair $(\Sigma, f)$ where $\Sigma$ is a state space and f is a possibly nondeterministic state successor function.

The state successor function f may possibly be decomposed into component successor functions $f_i$ where the $f_i$ are set functions. A component successor function $f_i$ may possibly be further decomposed into value successor functions $f_{ij}$ where the $f_{ij}$ are not set valued functions but are value space valued functions. Note that either the $f_i$ or the $f_{ij}$ may be nondeterministic. The definitions of the function decompositions are developed in the next section.

A computation of a process $(\Sigma, f)$ is a sequence $\sigma_0, \sigma_1, \sigma_2, \cdots, \sigma_i, \cdots$ such that $\sigma_i \varepsilon \Sigma$ $(i \geq 0)$ and $\sigma_{i+1} \varepsilon f(\sigma_i)$ $(i \geq 0)$ , and $\sigma_0$ is an initial state of the computations. Thus a process and an initial state define a computation.

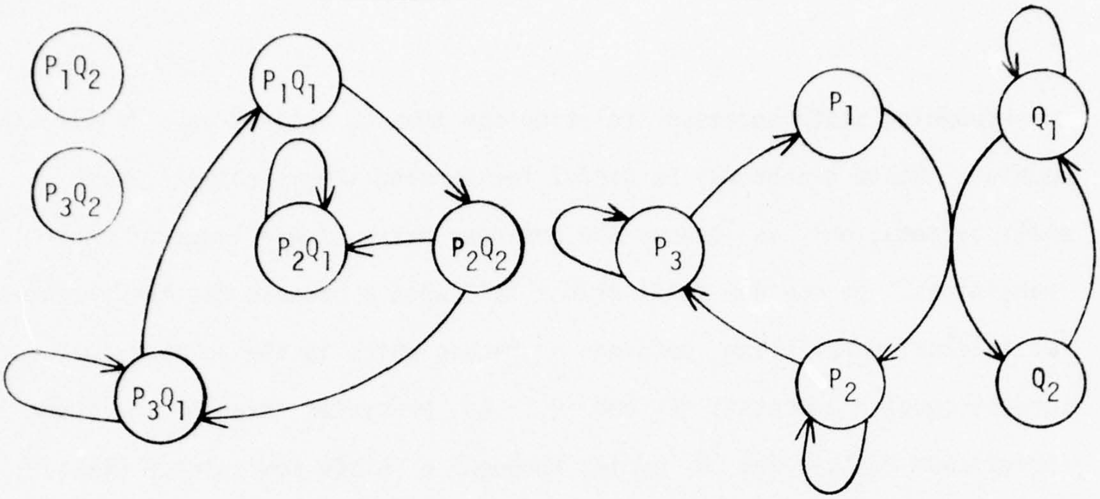
### 3.2.1 Process Graphs

Suppose that we have a successor relation f which we wish to decompose into simpler relations. Since only finite specifications of f are useful, writing a different relation for each single state will fail if there are an infinite number of states. The solution is to gather states into a finite number of equivalence classes and write a separate successor relation for each class.

If the equivalence classes are represented as nodes of a graph and the successor relations as arcs, the result is known as a state graph. The corresponding state successor relation can then be defined as a finite state machine. State graphs may be useful for forming specifications, even for small systems, only as long as the designer has a single locus of control transitions. To see how state graphs fail when a process has the potential for internal parallelism, consider a process which is the composition of two loosely coupled processes P and Q . As P cycles through m state equivalence classes and Q cycles through n state equivalence classes, the composite process will be cycling through a state graph of mn nodes. For each of the m possible values of component P , there will be a different variant of the component successor relation for Q . State graphs also fail because they quickly become unworkably complex as the number of state equivalence classes increases.

A state of the composite process could be represented as having two state components, one giving the state of P and the other giving the state of Q . This indicates a better way to graph the composite process: the nodes of the graph will be a finite set of equivalence classes of state components, and the arcs will represent successor relations on state components. This kind of graph will be called a process graph. The state and process graphs for the composite process are shown in Figure 8.

The process graph is a better characterization of what is going on in the composite process, and is much simpler, especially when m and n are large. Given the definition of the state successor function the two graphs are functionally equivalent. The state graph explicitly encodes information that the process graph only implicitly encodes, such as the fact that $P_1Q_2$ and $P_3Q_2$ are unreachable, for instance. However, this information is irrelevant to the problem of specifying f . If that information is relevant



(a) state graph

(b) process graph

Figure 8: State successor function graphs. $P_i$ and $Q_i$ are components of sub-processes $P$ and $Q$ respectively.

to some analysis, it may be either be obtained by studying computations via the process graph or it could not have been obtained in the first place. A formal definition of process graphs is given in Appendix A.

### 3.2.2 State Successor Function Decomposition

The decomposition of a state successor function $f$ into component successor functions can be represented by a process graph. Each node of the graph represents a state component space and each arc represents a component successor function $f_i$ . The arc is drawn from the component spaces in the domain of $f_i$ to the component spaces in the range of $f_i$ . For example, suppose we have the state successor function $f:\Sigma \rightarrow \Sigma$ where $\Sigma = \Sigma_1 \times \Sigma_2 \times \Sigma_3 \times \Sigma_4$ . Suppose further that $f$ can be decomposed into component successor functions $f_1$, $f_2$, $f_3$ given by:

$$f_1: \ \Sigma_2 \times \Sigma_4 \rightarrow \Sigma_1 \times \Sigma_3$$

$$f_2: \ \Sigma_1 \times \Sigma_2 \rightarrow \Sigma_2$$

$$f_3: \ \Sigma_3 \rightarrow \Sigma_4 \ .$$

Then $f$ can be represented by the process graph in figure 9.

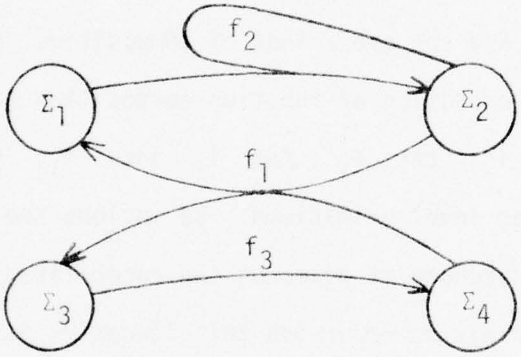


Figure 9: A process graph with component selector functions.

### 3.2.3 Functional Elaboration

The $f_i$ and $f_{ij}$ in a functional specification may be left as primitives or may be decomposed into lower level primitives. The functional specification must be based on primitive functions of the designer's choosing. These primitive functions may be arbitrarily simple or arbitrarily complex. The more complex the primitives are, the simpler the resulting specification structure will be, and the less help the designer will receive in analyzing it. The primitive functions must of course obey the design laws, in order to ensure the overall specification properties which the designer desires.

The ability to select these primitives freely lets the designer avoid the formalism if he wishes. He may elect to define f as a primitive, in which case there are no restrictions on it. Neither, of course, will be receive much help in analyzing it.

We must now decide what basic operations on functions we must have in creating the functional specification structure. From recursive function theory we know that all algorithms can be defined in terms of a few primitive functions and the operations of composition, primitive recursion, and selection. The operation of function composition must be included; with it we decompose f into the $f_i$ , the $f_i$ into $f_{ij}$ and high level primitives into lower level primitives. We include the operation of primitive recursion because it gives us the capability for iteration. By the definition of primitive recursion this iteration is bounded because the recursion is guaranteed to stop after a finite number of function evaluations. Finally we use the operation of function selection, which is defined as follows: $s(p_1:g_1,p_2:g_2,\cdots,g_k)$ evaluates to the value of the first $g_i$ such that $p_i$ evaluates to true (the $p_i$ are predicates which evaluate to true or false). This selector function s gives us a model of control.

It is worth noting that the basic form of a component function is that of a tree of nested functions with primitives as leaves. This form is established by the composition schema. Recursion and selection do not alter the tree form of the structure which is finally evaluated, but only delay its binding until evaluation time. Recursion finally expands to a fixed depth nesting, and selection simply reduces to the selected subtree.

At this point we can model an arbitrary algorithmic state successor function for a single system. We now need to introduce a functional model for interactions to deal with nearly decomposable component functions (decomposed with residual interactions) and with multiple interacting systems.

### 3.3 Interaction Specifications

We now define a class of primitive functions which will allow the designer to specify interactions. These exchange functions have the unique property that under certain conditions they will exchange values of arguments with a matching exchange function elsewhere in the specification. The exchange of arguments between a pair of matching exchange functions is accomplished by having each of them evaluate to the argument of the other. Exchange functions are labelled with subscripts and only exchange functions with the same label can match. The set of exchange functions with a given subscript is referred to as a class. Thus exchanges may only occur between members of the same class.

The three exchange functions XC, XA, XS are defined as follows:

$XC_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ which has been waiting for a matching exchange function

or

if this $XC_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$, $XA_i(\beta)$, or $XS_i(\beta)$ is evaluated.

$XA_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ which has been waiting for a matching exchange function to be evaluated

or

if this $XA_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$ or $XS_i(\beta)$ is evaluated.

$XS_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ which has been waiting for a matching exchange function to be evaluated.

$= \alpha$ otherwise.

### 3.3.2 Evaluation

Any state successor function can be defined by a definition tree as shown in figure 10a and automatically transformed into a corresponding precedence graph as shown in Figure 10b. The precedence graph simply displays the constraints on possible evaluation sequences. The use of exchange functions imposes additional (and potentially incompatible) synchronization constraints and allows values to be exchanged.

The exchange functions can be analyzed as normal (possibly nondeterministic) functions in their local context while still providing a high level (non-procedural) model for asynchronous conventional process specification to internally and externally asynchronous processes.

An internally asynchronous interaction could be defined as matching exchanges between component successor functions. An externally asynchronous interaction could be defined as matching exchanges between state successor functions (each defining an independent system). Thus all interface interactions are modelled directly and homogeneously by our functional specifications.

Using an immediate exchange, XS, we can also model what we will call unsynchronized systems as containing only XS type inter-system interactions. Such systems never wait on any interactions and are essential for many real time systems and for modelling the environmental system or real, physical world. An XS function cannot be used an an intra-system interaction since there cannot be sufficient constraints in a precedence graph to ever force its instantaneous matching with another exchange. Such use is therefore not allowed.

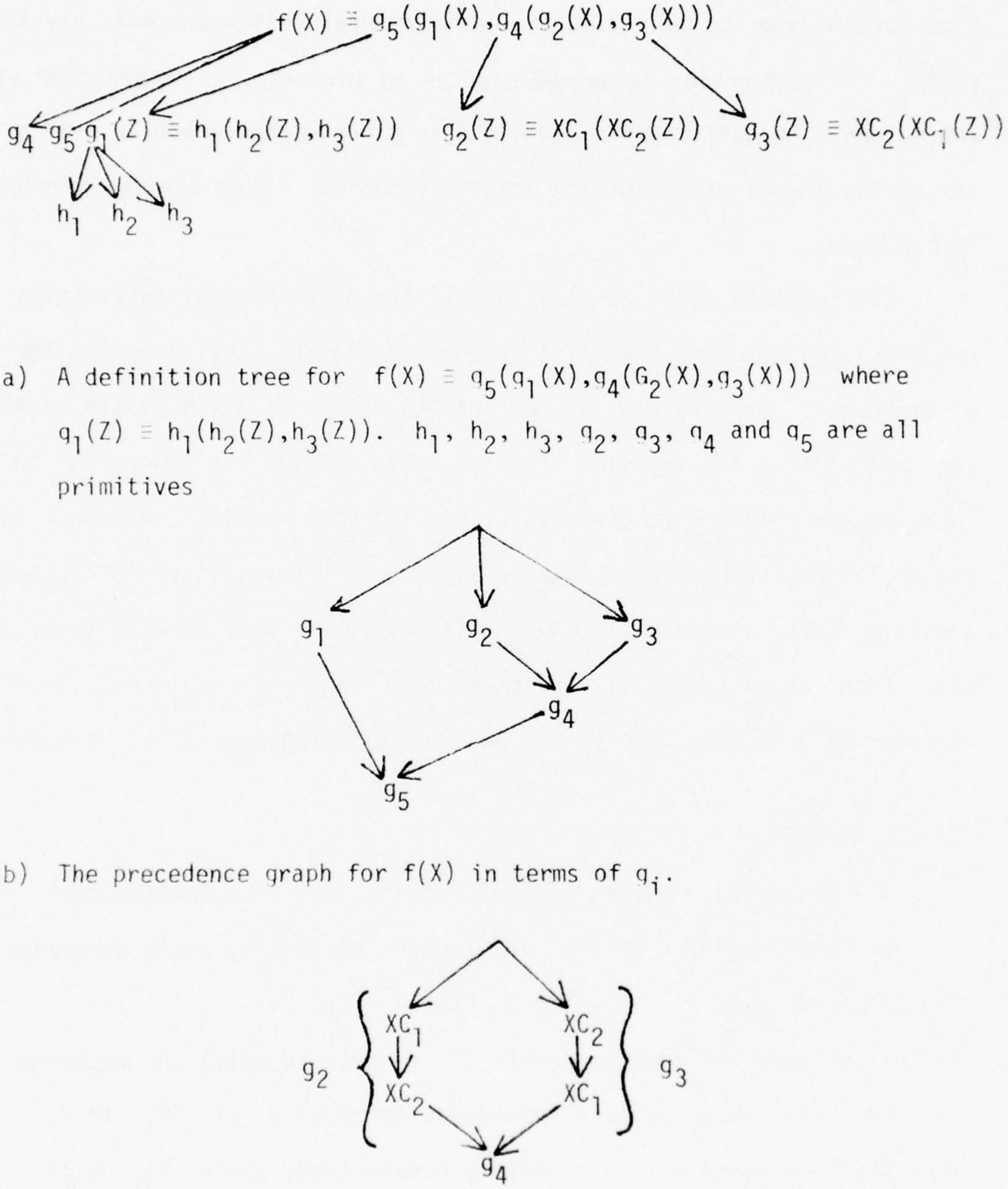
Unfortunately, this enormous generality of functional interaction specification comes at the price of some new design laws governing the use of exchanges. Arbitrary usage can lead to inter- or intra-system deadlocks (as must be true for any general interaction model). An example of an intra-system deadlock is given in figure 10c when no other exchanges of those classes are present. However it is possible to place restrictions on the form of the specification such that no process will be blocked in this way. Each restriction will correspond to a design law which must be followed in order to guarantee completion. Such restrictions are given in Appendix B.

### 3.3.3 Examples

A few trivial examples may help explain the use of exchanges.

We could define a pair of interacting systems by state successor functions f and f' as given in Figure 11a.

In this example the evaluation of f' is delayed until the exchange $XC_1$ has been completed by a subsequent evaluation of $XS_1$ in f. Thus the evaluation of f is not so constrained, since $XS_1$ will exchange with itself in order to continue without delay. f' could thus be interpreted as a system synchronized to the system f, that uses values from f in its own computations. The system f could be interpreted as a simple real-time clock that goes on with its

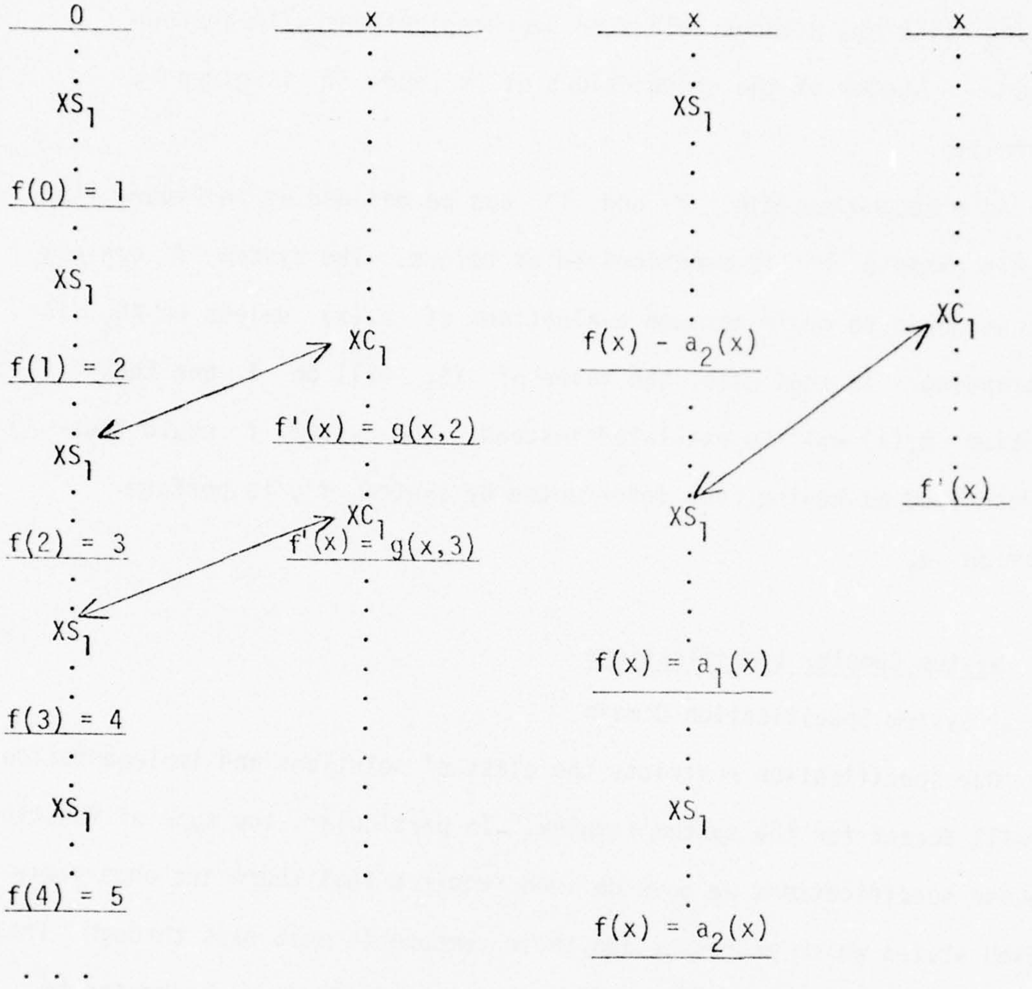


(a) A definition tree for $f(X) \equiv g_5(g_1(X),g_4(G_2(X),g_3(X)))$ where $g_1(Z) \equiv h_1(h_2(Z),h_3(Z))$. $h_1$, $h_2$, $h_3$, $g_2$, $g_3$, $g_4$ and $g_5$ are all primitives

(b) The precedence graph for f(X) in terms of $g_i$.

(c) A blocked precedence graph for $g_4(XC_1(XC_2(A)), XC_2(XC_1(B)))$ control cannot pass the first $XC_1$, $XC_2$ functions.

Figure 10: Use of exchanges in a function.



(a) Real time clock example

$f:\epsilon\to\epsilon \quad f(x) \equiv \text{First } (suc(x),xs_1(x))$

$f':\epsilon\to\epsilon \quad f'(x) \equiv g(x,xc_1(0))$

Where first $(x,y) = x$ and
suc $(x) = x+1$

(b) Interrupt system

$f:\epsilon\to\epsilon \quad f(x) \equiv (XS_1(F):a_1(x),T:a_2(a))$

$f':\epsilon'\to\epsilon' f'(x) \equiv g(x,xc_1(T))$

Where $T \equiv$ True and $F \equiv$ False

FIG 11: Simple Exchange Examples

cycling (ticking) without delays or synchronizations with another system. A sketch of the computations of f and f' is given in Figure 11a.

As a second example, f and f' can be defined as in Figure 11b. In this example f' is synchronized as before. The system f can now be considered to cycle through evaluations of $a_2(x)$ unless an $XC_1$ is outstanding. In that case, the value of $XS_1$ will be T and the function $a_1(x)$ will be evaulated instead. The system f could thus be described as having been interrupted by system f', to perform function $a_1$.

## 3.4 System Complex Specifications

### 3.4.1 System Specification Domain

Our specification restricts the class of solutions and implementations we will accept for the system complex. In particular, the type of functional process specifications we have defined requires that there are observable closed states which processes and their components must pass through . These states, which are needed for verification and testing, are generated by the constraints on events (function evaluations) which bind the synchrony of these events. However we may defer decisions about synchrony or asynchrony until their proper context by including the relevant events within a single process step, or if necessary by placing the events in different processes. Thus we may generate observable closed states at the levels where they are needed and defer other synchronizations. The necessity for placing events in different processes may arise from an unsynchronized components system (e.g. the physical world) or when there is no functional requirement for the relative rates of occurrences of those events, or no such requirement in the present design context. If relative rate requirements exist we may be able to include the events within one process, where desired properties are more easily ensured or decided. The flexibility which our functional specifications provide with respect to observed closed states and binding of synchrony is itself a desirable property.

Our process specification formalism is capable of specifying a large range of systems in a natural way. Its constructs reflect the ways we think about the systems under consideration, subject to the derived constraints which guarantee desirable properties of system behavior. Our functional framework lends itself to analysis of axiomatic constraints, yet within that framework we express the breaking of a state into components, the distinction between operations on sets and operations on values, and the decomposition of functions by the operators necessary for primitive recursive functions (composition, selection, and primitive recursion). Furthermore within the functional framework we express three primitive types of interaction, XS, XA, and XC. With these exchange functions we can express a wide variety of interactions. It is also important that we can express a wide variety of constraints on these exchange functions in order to ensure the integrity of asynchronous interactions; it is not unreasonable to say that asynchronous interactions create many of the problems in large distributed real-time systems. There is no generally accepted model of all asynchronous interactions, as there is of recursive functions, so it is difficult to claim that our exchange functions are completely general. However, they are adequate to model asynchronous and real-time interactions of existing computer systems.

### 3.4.2 Simulation and Testing

Good functional specifications allow simulation testing which is consistent and complete with respect to the behavior of the specified system. Our functional specifications can automatically be interpreted as a simulation model of the specified systems. Our formalism allows the stochastic simulation of both functional evaluations and relative occurrence times of events even without formal specification of primitive functions. A complete procedural model may also be generated when procedures for primitive functions are supplied. Simulating a functional specification is relatively easy, and can be done formally. The direct use of the functional specifications as the testable model prevents erroneous assumptions or obsolete models from invalidating the testing.

A very useful property, but one which can be ensured only by correct decisions in stating requirements, is the local testability of other properties of the specification. That is, we wish to be able to ensure or decide desirable properties by considering small portions of the total system specification. Finding axiomatic constraints which have an effect on this local testability would be an important area of future work.

### 3.4.3 Discussion of Other Formalisms

There are a number of functional specification formalisms which address some of the issues we have addressed. Chiefly among these are Petri nets, TRW requirement nets, and Higher Order Software (HOS).

Petri nets model system behavior by a network of events and conditions which relate the occurrence of events. The functional interpretation of events is done outside the network formalism. There have been extensive investigations of constraints on network specifications which ensure the absence of deadlock blocking of events or the efficient decidability of the absence of deadlock blocking.

TRW requirement nets model system behavior by a network of events, external interactions, and processing actions. Thus a requirement net is essentially a high level simulation of a system. Their formalism for process specification is well integrated with a developmental process methodology.

HOS specifies a system by a function which is decomposed into a hierarchy of mathematical functions according to a set of six axiomatic constraints. The primary impact of these axioms is in ensuring the integrity of functional interfaces and the existence of a controller guaranteeing that all functions in the hierarchy are evaluated. An interesting aspect of the work on HOS is in relating that functional process specification formalism to a formalism for developmental processes.

All of these approaches can be described in our formalism and compared with each other as well as with our functional specifications. This will be reported in our final report. Preliminary comparisons have been made, but a report on the results now might be inaccurate or misleading.

### 3.5 Summary

The formalism for functional process specifications must allow analyses for the properties needed by the requirements methodology. Furthermore, these properties are motivated by the specific types of systems required, that is real-time distributed data-processing systems. Some of these properties are listed in sections 2.1.2 and 3.1.2. Others are developed in following sections. In the case of certain properties, for example algorithmic implication and boundedness, we have indicated how the analysis of a functional specification may be carried out. Certain properties will be satisfied only if correct decisions are made in developing specifications; these include the locality of testing and the traceability of the impact of changes on the specification. Some of the properties are guaranteed by the definition of the formalism, for example generality and the ability to use high level primitives.

The high level (informally specified) primitives are especially important for the design process methodology, as discussed in section 2.5. High level primitives make it possible to factor requirements testing into well-defined, manageable steps. Primitive elaboration also constitutes a well-defined design step, and allows design decisions to be made in a local context.

The formalism we have defined allows the formal development of a requirements methodology for real-time distributed data-processing system. We have begun an analysis of properties of the formalism and have indicated areas needing further analysis. Although more design laws will be imposed, the functional formalism developed here seems a very suitable basis for identifying, studying, and solving the remaining problems.

We can now give useful formal functional specifications of networks (interacting complexes) of real-time systems and their real world environment. This can be done for any stage of the developmental process from requirements to implementation. Further, we may use the same functional formalism to study and define developmental process themselves.

# 4. REAL-TIME SYSTEMS

We now have a formal way to define interacting systems, and can address the questions of what a real-time system is, and how it can be tested.

## 4.1 Definitions

An independently-clocked system has only intrinsically non-waiting interactions with other systems. It is a system which, by its very nature, cannot wait for other systems, i.e., it does not wait. It will continue to run as the real-time clock runs: the missiles will continue their flight even if the data-processing system is deadlocked; the radar signals will move through the air regardless of who is listening. These systems will have only XS interactions. A partially synchronized system is one that executes inter-system exchanges that are XC or XA (not XS) types. Such a system cannot be independently clocked or measured.

The relevant characteristic of a real-time system is a bounding on the path from a given stimulus to a given response.[Ph 76] The bounding is necessary because the response will have a direct or indirect effect on some independently-clocked system. (If the effect was on a system that did not have to be independently clocked, that system could simply wait until the response was given, i.e., it could synchronize with the responding system). For example, the responses might be to change the setting of some variable in a nuclear reactor, or to maneuver an interceptor

[Ph 76] Phillips, Jorge V., and Bredt, Thomas H., "Design and Verification of Real-Time Systems", Proceedings of the 2nd International Software Engineering Conference, San Francisco, California (October 1976), pp. 124.

aimed at an incoming missile. The bounding is important because these systems are independently clocked and therefore do not wait. If the response takes too long, it may be too late to keep the reactor from exploding or the missile from reaching its target. (Note that the critical factor for the definition of real-time is the time bound required for interaction with an independently-clocked system; by this definition an on-line interactive airline reservations systems would not be considered real-time, but an air-traffic controller would be). The bounding is necessary (if not bounded, there is no minimum evalution speed sufficient to meet requirements) but not sufficient (the implementation may not meet the minimum speed required).

The designer must specify which paths must be bounded; not all interactions with independently-clocked systems are critical. The designer must also indicate how critical each path is, as a distribution function or probability that a given time bound will be reached. If it is imperative that the time bound always be reached, then the bound is required 100% of the time. However, decreasing the probability may permit significant improvements in efficiency. For example, worst-case allocation of resources may not be required; a very good heuristic may be used in place of an expensive algorithm.

If one need not operate at 100% probability, that relaxation gives the designer an additional flexibility. However, it also imposes certain constraints:

1) The probability must be figured accurately and reliably.
2) One must be able to know when the time bound has been exceeded.
3) One must specify appropriate recovery action to be taken if the time bound has been exceeded. The "recovery" may be only a

message ("Whoops! Chicago destroyed. Deepest regrets.") or an intricate procedure to continue as best as possible.

## 4.2 Path Bounds

To show that the path from the stimulus to the response is bounded, one must first of all be able to define that path, and then to prove that the path is bounded. Obviously, the complexity of the path will determine the ease with which this can be done. The following classification of kinds of paths is in order of increasing complexity.

1) If the response to the stimulus is produced in the same system cycle in which the stimulus was received, then we know that the response path is bounded, if the state successor function is bounded. (Note that the path could be bounded even if the state successor function, while algorithmic, is not bounded, but the proof of boundedness would require additional analysis.) Our formal functional specification makes it easy to decide if a function is bounded, since building functions by using only the operations of composition, primitive recursion, and bounded subtree selection will intrinsically give a bounded function. Our formalism provides sufficient (though not necessary) conditions for boundedness; the formalism can thus be considered as a design tool for proof of boundedness on this level.

2) Inter-step but intra-system computations, however, are not intrinsically bounded or even algorithmic; these properties must

be proved to exist. A simple path without loops and without interactions would easily be proved to be bounded. However, the existence of loops would require an analysis for a bounding on the loop. Yet even the discovery of such a loop is not a trivial matter. For example, in bounded subtree selection, the first predicate could be used as a loop-exit condition, where its corresponding value could be the response or could trigger the response in some other function. Alternatively, the third predicate might indicate loop exit, in which case the negation of the first two predicates as well as affirmation of the third would be required for loop exit. These loop-exit conditions, as simple or as complex as they may be, could be used as induction variables. The value of the induction variable would cause the loop to be exited at a given point, namely when that variable takes on a certain value or range of values. If it can be proved that a traversal of the loop brings this variable closer to the exiting value, then by induction one can prove that the loop is bounded.

One may want to have design laws to control the complexity of the induction variable, as well as to make the induction variable easier to identify. If automatic discovery of the loop and its exiting conditions proves to be too difficult or too computationally complex, the designer may have to specify the loop and exit conditions, perhaps using automatic verification of the specified paths as a design tool.

3) When the path involves inter-system interaction, more complexities are involved, for now at least part of the whole complex must be analyzed. For example, if our original system does an inter-system XC[i], it will wait until some other system does an inter-system XS[i], XC[i], or XA[i], unless an inter-system XC[i] or XA[i] is already outstanding. One must prove that if the latter is not true, then some other system will execute an inter-system XS[i], XC[i], or XA[i] within a bounded amount of time. The levels may go deeper, of course: the inter-system XS[i] from system A may not be executed until an XC[j] is satisfied by an interaction from system B, etc. For the purposes of this inter-system interaction analysis, we can form precedence graph abstractions of the system in which only the inter-system interactions are relevant. A design law requiring that intra-system exchanges have different indices from inter-system exchanges would allow such an abstraction to be made. The designer could then work with these abstractions to determine if a given interaction would always complete, i.e., would be mated with another interaction, within a bounded amount of time. See Appendix B for further information on such mating theorems.

We also have other options as designers. The case 2) specification may be given instead as a case 1), with initial to final state step occurring in only one state successor function evaluation. Some case 3) specifications may also be given with interactions between independently-clocked systems as similar "single step" synchronized system specifications. Much more work will be required on this topic.

## 4.3 Non-real-time testing

Simulation is one of the chief means of testing large complex systems. Since the actual timing is so important in real-time systems, they have traditionally been tested by real-time simulation. However, the real-time environment which must be simulated may get so complex that it can no longer be computed in real-time. One would then want to do non-real-time testing, knowing that a successful test in a "slowed-down" real-time environment would imply a successful test in a real-time environment running at normal real-time speeds. What constraints on the system are necessary so that such non-real-time testing will be valid?

First we must establish the level of the system being tested. Presumably prior analysis of the functional specifications affirmed that the stimulus-response path is bounded. Knowing that the path is deterministic and bounded independent of the relative speeds of systems already is very useful; it also means that real-time tests need only to look at path times and correlations. The simulation tests would be run on the actual implementation, since that is the level at which physical timing measures are meaningful. The difference between real-time and non-real-time testing is the speed at which the stimuli enter, i.e., the interarrival times. At the functional level this speed may not be relevant; it may have been abstracted out. If there are functional performance dependencies, they can of course be tested in a non-real-time way. However, at the actual physical implementation level, the interarrival times do affect the actual times for path traversal. Thus a virtual operating system with an infinite number of virtual resources would show no effect, but a physical operating system with a bounded number of resources would impose inter-performance dependencies via physical contention for resources. Different physical resource management polices would give different dependencies and thus different timings.

The relevant factors for determining the effect of the timing include the number of virtual resources needed, the number of physical resources available, and the resource management policy which couples the two. Given these three factors, if one could prove that the rate of path traversal was not affected (or how much it was affected) by the number of paths being traversed (i.e., by the number of stimuli being processed at one time), then non-real-time testing should produce the same results as real-time testing. Perhaps analysis could also show the point at which non-real-time testing is no longer reliable for a given resource management policy (increasing the number of virtual resources required or decreasing the number of physical resources available could push a system past the threshold of reliable non-real-time testing). Such analyses would also give measures for evaluating different resource management policies. In the limit of light loading, physical resource contention has minimal effect. In this case the only real-time testing required is to find maximum loadings for given path performances. All other testing could be done non-real time.

Note that this analysis relies on the proof that the rate of path traversal is not affected by the number of paths being traversed. Currently, there are no design laws which would make such a proof possible, nor even any strong indications that such design laws do exist in a form which is not overly-constraining. This is simply one possible approach to the problem.

Another approach involves taking measurements to determine path correlation. Thus one could measure the performance times on each path as a function of the stimulus frequency. Then one could determine the effect on performance when paths are coupled; this would show the effects of contention. From this data one could develop a model for the coupling between paths. This model could then be used to extend the results of non-real-time simulation testing to expectations of real-time testing results. Only real-time experiments to measure such correlations would be required.

## 4.4 Summary

We have distinguished between independently-clocked systems, synchonized systems, and real-time systems. The functional requirements for real-time systems (as opposed to non-real-time) consist of performance distributions of specified path traversals. Much of the required testing can be done functionally. It may be possible to restrict real time testing to either (or both) of

- Maximum load tests
- Path interference measurements

and allow a systematic and minimal real-time test program to be designed. Much work is required in this area and only a preliminary set of critical issues have been defined.

# 5. DISTRIBUTED DATA PROCESSING SYSTEMS

The requirements specification process was discussed in section 2. Section 5 will present an example of some of that section's ideas concerning system (and requirements) decomposition/integration, in the context of the design of a distributed data processing system. It should be noted that the system presented in this section was designed before many of the concepts discussed in Section 2 were formalized and in fact served as a motivating force for some of that work. Thus the purpose of the inclusion of this example in this document is not to claim that this is the "best" distributed data processing system but rather to illustrate the effect of some of the ideas of Section 2 on a particular design process.

## 5.1 Introduction

Distributed data processing systems encounter problems which are unique among computing systems. Not only are the problems common to large scale computing systems present, such as the effective use of system facilities by the users of the system and software redevelopment in response to changing technology, but also the problems created by an often very large class of users with widely differing application needs. Problems also arise from the desire of users of a network to communicate with each other to an extent not found in conventional systems, in order to gain information or share resources. With the added probability of errant or malicious processes, it seems that the conventional centralized operating system is inadequate. Thus the network designer is forced to look toward a decentralization of the network operating system in order to solve his problem.

This section will discuss an example of such a decentralized network system which was originally presented in [KR73]. The discussion will deal with both the design process and the design itself. Section 5.2 will discuss the design process of the network as an example of some of the system decomposition/integration ideas presented in Section 2. Also in Section 5.2 will be discussed the design constraints postulated prior to the design and their effects on system decomposition/integration. Section 5.3 gives a brief description of the network itself and discusses some design decisions as they relate to Section 5.2. A more complete presentation of the network of the design is presented in Appendix C. Section 5.4 will deal with the generality of the resulting network, and Section 5.5 will summarize the discus_ion of Section 5.

## 5.2 System Decomposition/Integration

As discussed in Section 2, in order to avoid the requirements allocation problem, a system must be factored in such a way as to allow the logical, resource, and performance requirements to also be factored over the sub-systems. As was pointed out, the requirements allocation problem can be minimized by identifying the "tightly coupled" attributes of the system and factoring the system with respect to these. It is also necessary to retain in each factor an abstraction of the entire system which indicates to the designer of the sub-system how his sub-system fits into the entire system. In some sense one can imagine a factoring of a distributed data processing system into 1) Application system, 2) Operating

[KR 73] Kramer, J.F., A General Structure for Uncooperative Processes Distributed Over a System Network. Ph. D. Thesis. Madison, Wisconsin: University of Wisconsin, 1973.

system, and 3) Hardware system. It is not clear at this stage, however, that a reasonable factoring of the requirements can occur over these subsystems.

To ensure such a requirement factoring, Kramer decomposes the operating system into two operating systems, the virtual operating system and the physical operating system. He now can present to the applications designer an abstraction of the system in which the designer sees only the virtual operating system and can specify his processes in terms of virtual resources and virtual operating system primitives, without any concern for the hardware implementation of the system. Similarly, the hardware designer's view of the system is application-independent: his only concern is to implement the physical operating system. The physical operating system designer's problem is now also clearly defined. He must map the virtual resources onto the physical resources. Since all three designers have nearly independent views of the system, minimal constraints are placed on them in their desire to optimize their sub-systems to their own performance, resource or logical requirements.

The method of specification which the applications designer uses to specify his processes can be that which was described in Section 3. The same is true for specifying the virtual operating system operations. An example of this type of specification for the Kramer network system functions is given in Appendix D.

The system integration phase can occur when the applications designer has specified his processes in terms of virtual operating system procedures which can then be compiled or translated into physical operating system procedures. The physical operating system procedures can then be compiled or translated into machine code which can be executed. This process is illustrated in Fig. 4 of Section 2. It is not clear at this point whether system integration must take place in this manner or whether something on the order of a purely functional specification of the application process can be "compiled" into some specification of the physical operating system.

Even though a comparatively large amount of freedom has been given to the individual designers to optimize their own sub-system, there is no guarantee that all the requirements will be met on system integration, thus the possibility of a design feed-back loop.

Once Kramer has established the factorization of the system, he postulates the following design constraints:

A. Network Communication

"The designed network must consist of a set of interacting, bounded, programmable, digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes."

B. Responsibility factorization

"Responsibility for meeting the postulated design constraints and any design decisions must be factored between the network, implementation, and process designers."

C. Authority

"Each designer must have sufficient authority to define the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility."

D. Delegation

"Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains the responsibility for that interaction and the authority to control the proces to which it was delegated."

E. Modification

"The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process managers all modification decisions and authority to control the effects of such decisions on the processes."

The design constraints fall into two categories:

(1) Those constraints which ensure some desired network characteristics: constraints A, D, and E. Constraint A requires that the network integrity must be maintained in terms of its own processes and its interactions with foreign processes. Constraint D requires that no design decision is made which arbitrarily centralizes authority, thus ensuring a maximally decentralized network. Constraint E provides a means for network evolution to meet future needs.

(2) Those constraints whose only purpose is to preserve the decomposition obtained at the beginning of the design process: constraints B and C. Constraint B states that each sub-system designer will have the responsibility to meet the other de-

sign constraints as they apply to his sub-system and Constraint C requires that each sub-system designer will in fact have the authority to do this. As an example, it would be a violation of the decomposition if the applications designer could specify as part of his application exactly how the virtual memory he used was to be mapped onto physical memory, as this is under the realm of the operating systems designer. Consequently this action also violates Constraint C.

### 5.3 Brief Overview of the Network Design

This section will present a condensed version of the design of the network presented in Appendix D, after which some of the design decisions will be discussed in terms of the design constraints presented earlier.

#### 5.3.1 System Structures

If the design constraints stated in Section 5.2 are to be useful, some formal definitions are necessary. A digital system is composed of two parts, a system processor and a system state. Operators are applied by the system processor to define a new system state or to construct messages for transmission to other system states in a set of digital systems. The execution of a system processor occurs in two distinct phases. Given an initial system state, the processor evaluates all operators which apply and produces a new local system state. Each operator can be executed in parallel and without side effects on the other operators being executed. The local system state is then updated by any messages from other system processors, and the local interpretation cycle begins again. This sequence is called a *system step* and transforms the system state into its successor state. This discrete sequence of states is called a *digital computation*. A set of digital computations represents a *digital process*. The interpretation cycle of each system in a set of interacting digital systems is independent and asynchronous. All inter-system communication will be defined in terms of asynchronous message transmission with any desired synchronization explicitly carried out by the cooperating digital processes. Messages will be received, after a finite delay, in the same order as they were transmitted by a given system. Some particular digital systems are defined as network systems, and a set of such systems as a network. There is a universal simulator of such sets of digital systems. Thus the system specification is already in a form that can be studied and tested. There are no physical implementation constraints on how many network systems may be implemented on a single physical system or on how many physical systems are used for a network or network system.

The network system states are defined as a set of elements. Each element can be interpreted as the state of a synchronously interacting process component of the overall system process. All system state information and all system computations are based on these system process elements. Network system operators are constrained so that at most one one operator will modify a given system state element in a given interpretation step.

Although we wish our system processors to be well-defined with respect to their operations on the system state, we desire also to permit them to communicate with systems outside of the formally defined networks which are not well-defined. Such systems are called foreign systems and are not constrained in their internal structure by the network designers, except for a communication conventions they must use if they wish to interact with a network system. Certain elements of the system state are housekeeping processes that manage inter- and intra-system interactions. These elements contain system state information and are neither explicitly transformed by application programs nor transportable between systems in the network. Those elements of the state that are explicitly programmed, transformed, and moved from system to system will be called network (i.e. user) processes. A network process step consists of a cycle of three system process steps or phases. Two of the system phases are used for housekeeping functions and will be described later. Operators applied during phase two of a network process step will be called system subprocessors; their effect is local to that network process state. The complexity of a given network process step is therefore defined by the corresponding subprocessor. The process state defines the current address space of the process. Figure 1 in Appendix C is an informal characterization of one network system.

In review, a foreign system is not characterized (or constrained) except as a source or destination of messages. A network system is composed of a system processor and a system state. A system processor is composed of a set of operators, some of which are subprocessors.

The system state is composed of system housekeeping elements and system elements interpreted as network (user) process states. A network consists of an inter-communicating set of such systems and foreign systems.

### 5.3.2 Network System Interactions

The mechanism for inter-system communication selected here is quite similar to the notion of a hierarchical interrupt system. In phase one, the system processor determines which sub-processors (if any) should be applied to each network process state in that system and delivers the appropriate messages (if any) to an interface buffer in each network process state. In phase two, the system processor applies the selected sub-processors to the selected network process states thus causing them to undergo a network process step. In phase three, the system processor performs the non-local services requested (if any) by messages left in the interface buffer by a sub-processor. Non-local operations are those which have side effects external to a network process state. Messages may be received by the system processor in all phases and are buffered until their delivery in some subsequent system phase one.

The system processor must be able to recognize, in any network process state, which sub-processor to apply in phase two, and each sub-processor must be able to recognize its own state information component in that network process state. For these purposes, each network process state will contain one or more objects called control points. Although control points are used for a variety of purposes, only those aspects affecting sub-processor application will be described here. A control point roughly corresponds to an interrupt level in conventional systems.

All control points in a network process state are ordered by priority within that state. Each control point has an ordered set of channel terminations and buffers associated with it in the system state elements for the receipt of messages. Depending upon the status of the control point buffers and also the status of the control point within a process state itself, messages may or may not be received by the buffers , and the control point may or may not be eligible for sub-processor allocation during a subsequent phase two. The details of message delivery are covered in Appendix C. The highest priority control point candidate present in a network process state will be made active , and the pending message, if any, will be placed in the process interface buffer.

During phase two, the appropriate subprocessors are applied to network process states containing the active control points selected during phase one. At most one subprocessor is applied to a given process state during any phase two step. Although many process states in a given system may be requesting the same subprocessor, they each have it applied in parallel during phase two. Whether they are really serviced in sequence or in parallel is an implementation decision that affects only the duration of phase two since no interprocess interactions in this network system can occur until phase two has completed. All processes containing active control points will have sub-processors applied prior to the end of phase two.

Each sub-processor transforms only the network process state to which it is being applied. The applied sub-processor thus defines the successor network process state. If a system service is requested (a non-local transformation), the sub-processor will complete its network process step leaving a message requesting the service in the local interface buffer for subsequent phase three processing. Note that network processes in the same system will proceed in synchronous parallel with each other, each completing one process step in each system cycle. Network processes in different systems will run asynchronously parallel.

During phase three all requests for non-local services which were left in the interface buffer are acted upon. There are four types of these services: message transmission, resource transmission, process state transmission, and system modification. Phase three ends when all such services have been completed and the interface buffers are cleared.

If the interface buffer contains a request for message transmission, the associated message is removed from the buffer and "broadcast" to all systems but addressed to a particular control point buffer. The system containing the control point (destination) buffer will receive it, other systems will ignore the broadcast. See Figure 2 in Appendix C. The message has a network standard format and is represented as a string over a network standard character set.

Although messages broadcast during phase three are subject to unspecified transmission delays (unless messages are intra-system), the order of transmission between any two systems is preserved in perception. In the receiving system, messages are placed in channel terminations buffers and used to update control point buffers during every phase. As far as network processes are concerned, message transmission is transparent to system boundaries, and network processes may cooperatively move from system to system without losing communication or restricting their interactions.

A network process defines the local environment and address space for sub-processor transformations. The interface buffer serves to factor the sub-processor transformations which are local to the process state from the system processor services which have effects outside of the process state. A process state will contain one or more control points, the status of which may be modified during system phase one as a result of message delivery or in system phase two as a result of control points in interprocess communication. Control points also serve to delimit uniquely a portion of the network process state, called an expression state. Within this part of the process state the designers of the corresponding sub-processor are responsible for defining both the representations and the transformations which that sub-processor will carry out on those representations. All of the state information required for a sub-processor to continue a computation must be part of any corresponding expression state. A network process state thus contains an interface buffer, a set of expression states and one or more control points.

Many freedoms are given to the network process designer involving complex process interactions, including manners in which to deal with uncooperative processes and to exploit known instances of cooperation. These are detailed more fully in Appendix C. In short, a single network process is capable of supporting all computations that can be carried out on a conventional single processor, multiprogramming system.

There is a sub-processor, called GP (General Programmable), in every network system. The GP expression state can be programmed, in the network system language, to provide on request all network services. GP expression states thus can specify invariant computations despite movement of the containing network process. The network system language thus plays many roles such as the following:

a) The network job control language.

b) The network high level "machine" language.

c) The network implementation language for operating "systems".

d) The base language for definitional extension via source language macros or compilers.

e) The base language for evolutionary augmentation.

f) The system invariant computation language.

The problem of deciding when a given interaction is improper can only be resolved by another process aware of the interactions. The system can not resolve conflicting claims of one process with respect to the behavior of another process since such resolution may require intimate knowledge of the specific applications. In order to guarantee resolution there must be a responsible authority who can control and manage interactions between the warring parties. Our system imposes a hierarchy of uniquely designated responsible authorities in the form of a network process tree, as in Figure 4 in Appendix C, to make such an arbitration. All authority rests initially in the root of the tree. Although delegation to a child process is allowed, each root of a sub-tree remains responsible to its parent for the interactions of the processes in that sub-tree. An uncooperative root of a sub-tree will still be accountable to its parent process. To insure this accountability the network provides another sub-processor, the AO (Accountable Object) sub-processor. The details of AO are available in Appendix C.

The process tree can both grow, by creating new "leafs", and shrink, by destroying "leafs". An existing process may move, as permitted by the parental authority, to any network system known to that parent. Thus both the process tree and its distribution over the network systems can change dynamically as a result of process computations. Since these are intrinsically non-local operations, the GP sub-processor can only request them. The responsibility for carrying them out has been delegated to a PR (Process Receive) process and its corresponding PR sub-processor. Each non-foreign network system will contain one of each. The GP requests for such service thus take the form of messages to the local (to the network system) PR process. Process birth and death and further details of the PR sub-processor are available in Appendix C.

### 5.3.3 Design Decisions

As was indicated in Section 5.2, allowing applications designers the ability to specify their processes in a hardware independent manner accrues many advantages, such as allowing hardware changes to take place without the necessity for rewiting applications, allowing a consistant virtual "machine" throughout the network which permits a process to move from one system to another to exploit special hardware features, and perhaps most important from a design standpoint, the network remains clearly factored into the sub-systems described in Section 5.2, allowing for local optimization to take place at all levels.

## 5.4 Generality

The system postulated in 5.2 and described in 5.3 seems general enough for a large variety of applications and, with the ability for self-modification, it would seem it could meet future needs. In addition to the generality it does allow for specialization in terms of special purpose hardware systems.

The design process and the resulting network seem to indicate that many of the problems inherent in a distributed data processing system can be dealt with effectively in a decentralized network operating system environment.

## 5.5 Summary

The design process of the Kramer system could be summarized as follows:

1) Decompose the system into subsystems over which the requirements can be factored. This necessitates the division of the operating system into a virtual operating system and a physical operating system.

2) Postulate design constraints so that subsequent design decisions in all sub-system will

   a) preserve the external requirements, i.e. well-definedness, provision for evolution, decentralization.

b) preserve the decomposition obtained in 1.

3) Develop the design or specification of the sub-systems to the point where integration can take place, i.e. the translation of application processes into virtual operating system processes and again into hardware processes, as illustrated in figure 4 of Section 2.

Once again, to the extent that (1) is accomplished without side effects, (3) will be correspondingly easier. This design process allows for local optimization of sub-systems as well as local testability, in the sense that application processes can be tested on a "virtual" machine independent of implementation, and the mapping of virtual resources onto physical resources can be tested in the absence of particular application constraints.

Our functional specifications can describe the resulting "Kramer Systems" and processes. The need for decomposition into application (again decomposed if needed), virtual operating, and physical operating systems lead to additional "good design" principles for requirements specification, i.e. state requirements in terms of such models.

# 6. THE DESIGN SCHEMA

## 6.1 Introduction

In this section we deal with the methodology of system design primarily in the context of the formalism developed in the previous sections. Yet the present discussion takes place on a plane of abstraction somewhat different from the earlier one since we are interested in a general strategy, more formally a design schema, to be realized as an advanced set of tools for the systems designer. Our development of this strategy is intended to be as unbiased as possible with respect to the particular aims of a given designer or to the ultimate application of the systems which he specifies. Further, we want to restrict ourselves to design techniques which maximize the possibility of local analysis and testing of system modules throughout the course of a top-down system definition. It should be noted that the development of our new design tools will depend upon some as yet underived consequences of the formalism of earlier sections. Thus our purpose here will be to explore those properties which we are already prepared to require of the proposed design package in order to identify the directions of future research in this area.

## 6.2 Design Processes

A key concept in this discussion is that of the design process, which is the second state in the developmental process. In the most general sense we can define a design process as a sequence of analyses, decisions, and commitments (all design steps) which start with a formal requirements specification and lead to a formal system specification implementation. As discussed in section 2, there is substantial overlap between the requirements process (which may also carry out part of a design process) and the design process itself. Much of the discussion of the formal requirements process steps in section 2 is directly applicable to this section. Obviously there are many design processes which can lead to a single product, and many products which can satisfy a given set of system requirements. Moreover we may characterize these design processes grossly in terms of such factors as cost, total human effort expended, and success of the product in meeting requirements. However, it becomes apparent thatwe must find it difficult indeed to model a design process after the fact if the sequence of analyses and decisions mentioned above is poorly structured, for example, as with decisions made by the designer with an uncertain or imprecisely stated impact on future decisions. We could (and in current practice usually do) have design decisions which are recorded only as informal statements or which are presented in no tangible artifact at all. Furthermore, given poorly structured design techniques, if we choose to restrict the evolution of design processes in order to control some function of the contingencies of the design steps (such as cost or time incurred in executing the product software), then we face an even more complex problem.

Since our goal is in fact to provide design laws which will allow us to satisfy system requirements, we are inevitably led by the arguments of the preceding paragraph to demand that steps in the design process be well-defined, that is, that termination of a design process step can be determined by criteria imposed by the design laws and systems requirements. Viewed in this respect the design process shares features with the computational processes described in earlier sections. In particular, we have functional specifications as the states of the design process, transitions between these well-defined states (carried out by the designer-aided digital computations), and finally possible interactions between asynchronous design processes.

The major difference between design processes and the processes specified in section 3, apart from the difference in state values, is that in the former processes state transitions, or design steps, cannot be guaranteed to be algorithmic (finite) or bounded in computation time. This inability to bound computation time results from the potentially complex nature of many design steps, for which no effective procedures exist. Further, human designers can carry out design steps that no machine can do and are not absolutely required to succeed. A simple illustration would be an optimization step in which a design procedure may search in vain for an impossible optimization specified by functional requirements. Thus the need for human intervention in design steps makes a definition of these step procedures by strict mathematical algorithms unattainable. We summarize by saying that well-defined process steps, the termination of which can be decided, are a minimally acceptable framework for the conceptual and practical formalization of intended design laws.

### 6.3. Design Steps

Some new features which we wish to impose on design processes can best be introduced by first giving a review of the types of design steps which might occur within these processes. This cursory review is not meant to imply that to carry out a design step is a trivial matter. On the contrary, a great deal of design and computational effort may be involved. We have discussed already in section 2 the decomposition and integration of functional specifications and corresponding design processes; these will be elaborated in section 6.4. In the preceding paragraph we also mentioned briefly optimization steps, to be treated more fully in section 6.4. Interactions with other design processes, at the intermediate steps of a design process, must also be included to compensate for prior decompositions into loosely coupled requirements. The binding step corresponds roughly to implementation of some entity in terms of another, as in the binding of control to a particular sequence or in replacing a function by a procedure. In summary we have the following kinds of steps:

- Decomposition: factoring specifications
- Integration: composing specification
- Optimization: finding a better specification
- Interaction: inter-design process communication
- Binding: encoding a design decision by elaboration of detail.

We will require that our formal representation of processes and systems requirements be amenable to several types of analysis in conjunction with each of the kinds of steps above. In particular we want to know whether the design step is able to meet the requirements under the constraints imposed by previous design decisions. We also want to be able to subject the system to simulation or other investigation in order to proceed intelligently with further design decisions. However, in the event that a design step fails , we must be prepared to perform it again with a new set of parameters, or if that fails, to retreat to some earlier design step, repeat it with new parameters and move forward again. Finally, it is apparent that if design step computations, analyses, and simulations are mechanized then much of the cost and time required in systems design can be reduced. Also we can minimize the introduction of errors and maximize the ability to detect errors.

## 6.4 Decomposition, Optimization, Integration

In this section we will provide a sequence of design steps which might typically be applied to a simple functional specification in order to reduce it eventually to a set of procedures for evaluating the function originally defined. By way of example we will deal with a single expression which consists entirely of primitive functions, thus remaining at a single level of abstraction and ignoring at this time the more complex issue of hierarchical functional specifications. The three types of design steps which we encounter involve decomposition, optimization, and integration. Decomposition of functional specifications into modules (where we leave modules undefined for present purposes) is based here upon the ability to perform different component computations of expression evaluation in parallel, a key factor in subsequent optimization operations. Figure 12

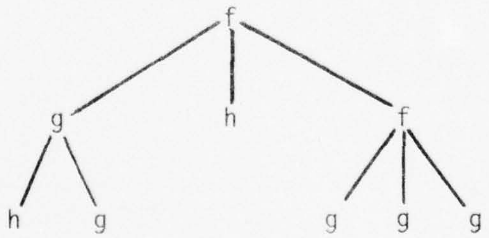
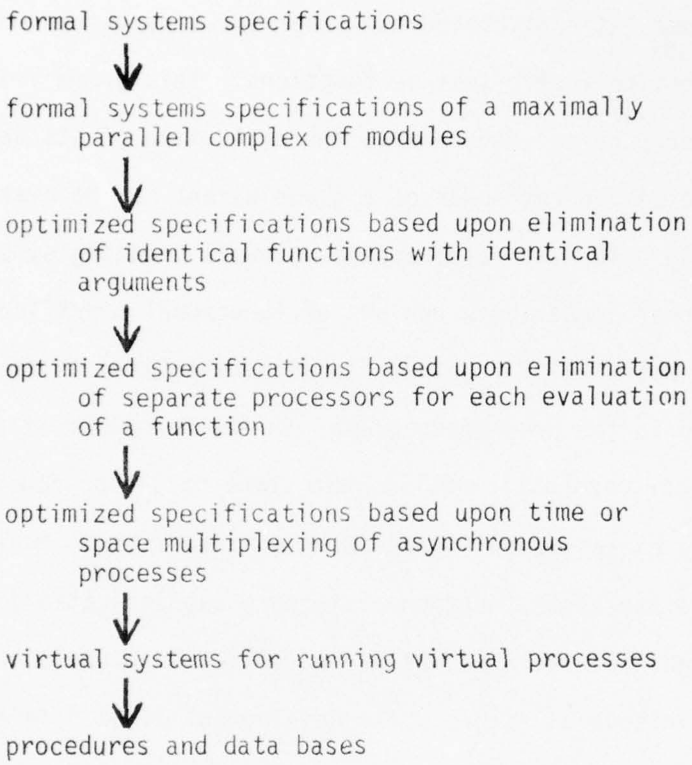


Figure 12. Precedence graph for the expression
f(g(h(a,b),g(c,d)),h(c,d),f(g(a,b),g(c,d),g(a,b)))

shows the computational precedence graph for a particular expression consisting entirely of primitive functions. This graph indicates that evaluation of a parent node awaits the evaluation of its descendent nodes , and that the descendent nodes of a given parent may be evaluated in parallel. An unsophisticated approach toward creating system modules (that is, toward creating a new set of functional specifications which reflect the potential for computational parallelism) is to assign a module to each node in the precedence graph. In the formalism of preceding sections, this would correspond to modeling the state successor function applicable to the expression of interest by an interacting complex of degenerate system state successor functions. (A more sophisticated decomposition approach might involve a recursive procedure which could be applied to the expression in several stages. The development of such an approach is a topic for further research.) We have thus described the decomposition process which comprises the first design step of Figure 13.

Our first type of optimization design step exploits computational parallelism in a simple way. It constitutes the second design step of Figure 13. For the sake of illustration let us assume that we are concerned with the evaluation of the expression f(g(a,b),g(c,d),g(a,b)), a subexpression of the one in Figure 12. Since the first and third arguments of the function f are identical, then we need only evaluate one argument and save the resulting value so that the same computation does not have to be performed for the other argument. Whether this optimization, or any other type to be discussed below, will in fact be performed depends on the functional requirements, since the optimization may increase the



formal systems specifications
formal systems specifications of a maximally parallel complex of modules
optimized specifications based upon elimination of identical functions with identical arguments
optimized specifications based upon elimination of separate processors for each evaluation of a function
optimized specifications based upon time or space multiplexing of asynchronous processes
virtual systems for running virtual processes
procedures and data bases

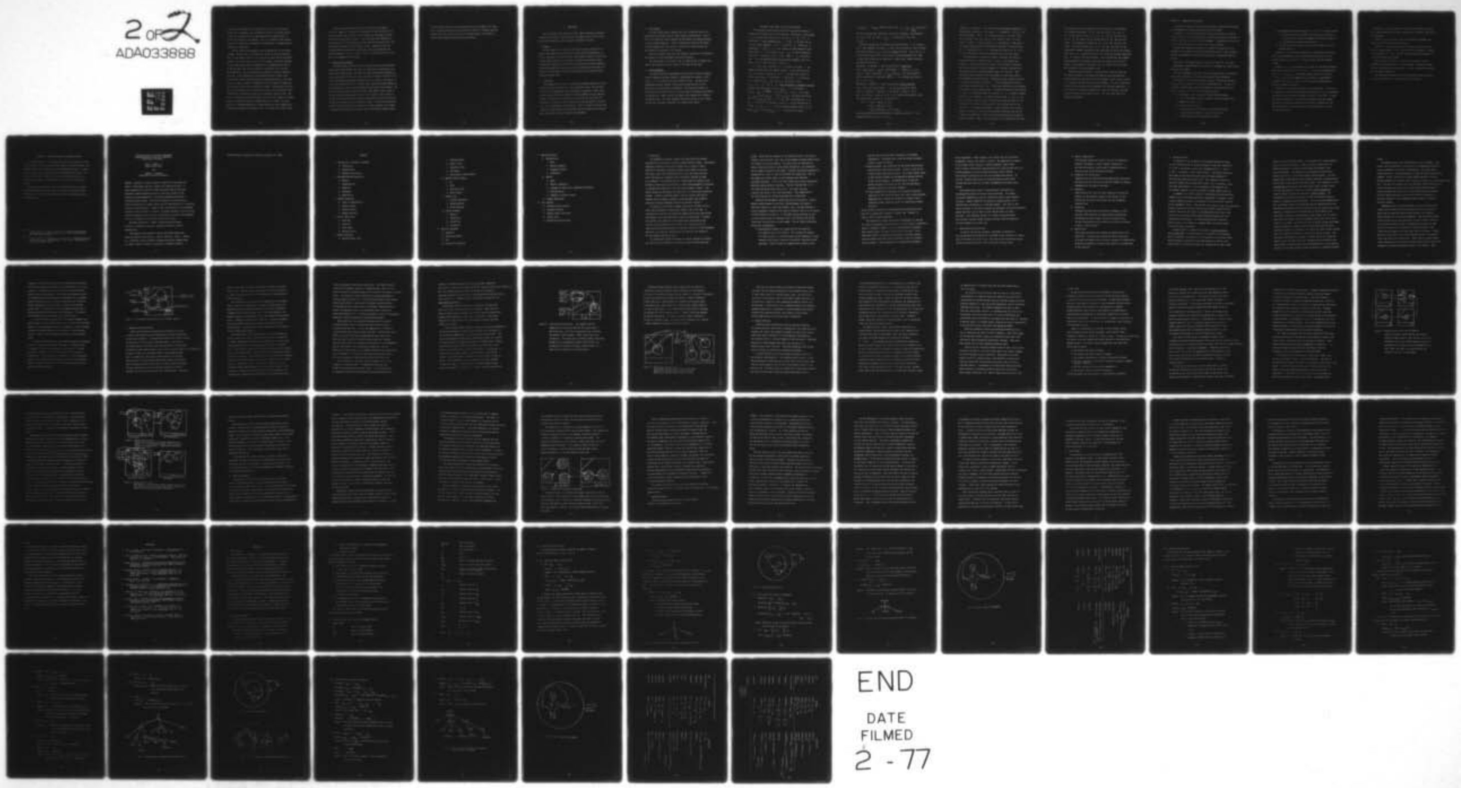
Figure 13. Steps in an elementary design process

computation time unacceptably. In the above example, however, the added complexity of saving the result would generally be exceeded by the cost of performing the evaluation twice.

The same expression which we used above also serves as an example for a second and subsequent type of optimization, namely elimination of separate processors for identical functions with different arguments, as g(a,b) and g(c,d) for the function f above. Here the savings in the number of processors needed for the computation is somewhat offset by the

AD-A033 888 WISCONSIN UNIV MADISON DEPT OF COMPUTER SCIENCES F/G 9/2
THE FORMAL DESIGN AND ANALYSIS OF DISTRIBUTED DATA PROCESSING S--ETC(U)
OCT 76 D R FITZWATER DASG60-76-C-0080
UNCLASSIFIED CSTR-279 NL

2 OF 2
ADA033888



END
DATE
FILMED
2 - 77

complication introduced in the state space as well as by some loss of parallelism. Presumably this optimization could be carried out mechanically, and the result could be demonstrated, again mechanically, to fulfill the functional requirements. This is an example of how analyses interact with functional requirements and design laws in determining the outcome of a design step.

We can include two subsequent types of optimization in our design strategy. Briefly, one is the storing of results of identical computations (in the object system) in one location for later use (by the object system). This necessitates the use of file managers and library managers in the system, which could be introduced in an automated way as part of the design step, thus avoiding additional responsibility on the part of the designer. The next optimization, potentially the most complex to carry out, is the combination of asynchronous parallel systems into a single system through time multiplexing or space multiplexing. (This type of step is clearly a kind of integration step also; however, integration steps in general need not be optimization steps.) The motivation for this last kind of optimization is obvious when we recall that functional specifications were initially factored into the maximally parallel form. This form is in general an unrealistic model for any but the simplest of systems because of the great expense of implementing such a system. The tasks entailed in automating methods for accomplishing this system composition, based upon the formalism developed so far, is a subject for further study.

Finally, we mention briefly the last two steps of the design process of Figure 13. These are the creation of virtual systems for the optimized system specifications and the creation of procedures and data bases within these virtual systems. These two steps thus complete the entire process of transforming the original functional specifications into procedures operating in virtual address spaces. However, we have not indicated how the procedures and data bases are to be mapped onto the physical hardware, since our design process is concerned with the logical part of requirements specifications.

## 6.5 Design Process Summary

Our main purpose in this section has been to show that well-defined design steps could minimize the time, cost, and error susceptibility associated with systems design. We have also tried to show that automation of design steps could relieve the systems designer of tedious responsibilities by providing detailed analytical information at each step in the design, of course with the possibility of iterating steps, and by providing feedback on the consequences of design decisions made interactively by the designer. We have seen many parallels between the formal requirements process steps of section 2 and the design process steps of this section. This similarity results from the fact that the design processes which we have discussed are the logical component of the requirements process. We have also mentioned several questions which must be addressed in future research as we attempt to elaborate and formalize the process steps of the design schema further.

In particular we have dealt only with design steps with regard to a single level of abstraction and have not explored the impact of exchange functions (within functional specifications) on the design steps or the impact of interactions between asynchronous design processes.

# 7. CONCLUSIONS

The preliminary study described in this report provides a conceptual framework within which a research program to address many of the critical issues can be developed with reasonable confidence of success.

## 7.1 Summary

We have identified a variety of different processes (each with its special constraints and associated design laws and transformations), an initial hierarchy of system properties (each based on aspects of real-time and distributed systems), and a formal functional specification system (meeting the constraints arising from the processes and system properties). A preliminary exploration of how to use these concepts in the design and analysis of distributed real-time systems opened up some new and highly relevant research areas.

## 7.2 Evaluation

The probability of success in meeting the research objectives of this contract has been increased to a very satisfactory level, and the plausibility of this approach has been substantially supported. There do not currently appear to be any potentially fatal weaknesses in this approach. It does not address all relevant issues in design, but those that are addressed are significant and of potentially very high payoff in terms of cost, time, reliability, and testability. There is reason to believe that extension of BMD problems to distributed real-time systems can be brought under intellectual control and effective management.

## 7.3 Final Report

The final report (due in January 1977) will extend the results reported here and present them in a more formally integrated way. We will not try to extend our results beyond the requirements and design processes or to new system properties. We will complete our survey of the issues, critical problems and their potential solutions for the processes and and properties introduced in this report.

We will also work up new (and more coherent) examples of our formalism and concepts in both requirements and design processes.

The final report will contain a set of research plans to address and resolve the remaining critical issues in the scope of this work.

## 7.4 Acknowledgements

The author gratefully acknowledges the assistance of many colleagues, former students and present students in developing the concepts in this report. It is not feasible to mention them all here. I do explicitly acknowledge the many valuable discussions and contributions of Prof. Pamela Zave at the University of Maryland. My research assistants at the University of Wisconsin have, of course, been of substantial assistance in preparing this report and are Tom Blumer, John Compton, Bill Hibbard, and Cynthia Hintz. Mr. James Rathmann volunteered to work on this project and has been a welcome collaborator in preparing this report.

## APPENDIX A-FUNCTIONAL PROCESS SPECIFICATIONS

The functional notation which we use for specifying processes is given by the following series of definitions. Definition: A value space $V$ is an unspecified primitive set. We use $V_i = V$ for $i=1,2...,n$. Definition: A state component space $\Sigma_i = P(V_i)$ is the power set of $V_i$, and a state component $\sigma_i \epsilon \Sigma_i$ is a subset of $V_i$. Definition: A state space $\Sigma = \overline{X}_{i=1}^{n} \Sigma_i$ is a cross product of state component spaces, and a state $\sigma \epsilon \Sigma$ is an n-tuple $(\sigma_1,\sigma_2,...,\sigma_n)$ of state components. Definition: A process is a pair $(\Sigma,f)$, where $\Sigma$ is a state space and $f$ is a (non-deterministic) state successor function which produces a state $\sigma'$ when applied to a state $\sigma$.

$f$ is much like a relation on $\Sigma \times \Sigma$ except its output $\sigma'$ may depend on interactions with another process. In the non-interacting case $f$ would be a relation on $\Sigma \times \Sigma$. In either case we need to define some notation for specifying $f$. Definition: $f$ is specified by an m-tuple $(g_1,g_2,...,g_m)$ where $g_i = (D_i,R_i,f_i)$ with $D_i,R_i \subseteq \{1,2,...,n\}$ and $f_i$ is a (non-deterministic) component successor function with domain $\mathop{\overline{X}}\limits_{k\epsilon D_i} \Sigma_k$ and range $\mathop{\overline{X}}\limits_{k\epsilon R_i} \Sigma_k$. $f$ is specified by $(g_1,g_2,...,g_m)$ as follows: $f(\sigma_1,\sigma_2,...,\sigma_n) = (\sigma_1',\sigma_2',...,\sigma_n')$ where $\sigma_i' = \bigcup\limits_{1\leq j\leq m} P_i(f_j((\sigma_k)_{k\epsilon D_j}))$. $P_i(f_j((\sigma_k)_{k\epsilon D_j}))$ represents the projection of $f_j((\sigma_k)_{k\epsilon D_j})$ onto $\Sigma_i$ if $i \epsilon R_j$, and is the empty set $\phi$ if $i \notin R_j$. Definition: A component successor $f_i$ may be specified by an $\ell$-tuple $(f_{i1},f_{i2},...,f_{i\ell})$ of value successor functions where $f_{ij}$ has domain $\mathop{\overline{X}}\limits_{k\epsilon D_i} V_k$ and range $\mathop{\overline{X}}\limits_{k\epsilon R_i} V_k$. $f_i$ is specified as follows:

$f_i((\sigma_k)_{k \in D_i}) = (\sigma'_k)_{k \in R_i}$ where for each $k \in R_i$, $\sigma'_k = \bigcup_{1 \leq j \leq \ell} \left( \bigcup_{Z \in \overline{X}_{s \in D_i} \sigma_s} P_k(f_{ij}(Z)) \right)$.

Similar to the previous definition, $P_k(f_{ij}(Z))$ represents the projection of $f_{ij}(Z)$ onto $V_k$. Note $Z \in \overline{X}_{s \in D_i} \sigma_s$ is a tuple $(Z_s)$ $s \in D_i$ with $Z_s \in \sigma_s \subseteq V_s$.

We have defined ways of decomposing a state successor $f$ of a process $(\sum, f)$ into component successors $f_i$, which are set functions, and if desired into value successors $f_{ij}$, which are element functions. The $f_i$ and $f_{ij}$ in a specification may be left as primitives or may be decomposed into trees of lower level primitives by operations of composition, subtree selection, and primitive recursion.

Definition: A function $f$ may be specified as a composition $f(x) = h(g_1(x), g_2(x), \ldots, g_k(x))$ where domain $f$ = domain $g_i$, $1 \leq i \leq k$, range $f$ = range $h$, and domain $h = \overline{X}_{1 \leq i \leq k}$ range $g_i$. This may be applied with non-deterministic functions, and for $f$ in a decomposition tree of an $f_i$ or an $f_{ij}$.

Definition: A function $f$ may be specified by subtree selection $f(x) = (g_1(x):h_1(x), g_2(x):h_2(x), \ldots, g_{k-1}(x):h_{k-1}(x), h_k(x))$ where domain $f$ = domain $g_i$ = domain $h_i$, $1 \leq i \leq k$, range $f$ = range $h_i$, $1 \leq i \leq k$, and $\{$true, false$\}$ = range $g_i$, $1 \leq i \leq k-1$. This specification models a flow of control which, in terms of if-then-else, is

$f(x) =$ if $g_1(x)$ then $h_1(x)$ else
if $g_2(x)$ then $h_2(x)$ else
$\vdots$
if $g_{k-1}(x)$ then $h_{k-1}(x)$ else $h_k(x)$ .

This may be applied with non-deterministic functions, and for $f$ in a decomposition tree of an $f_i$ or an $f_{ij}$.

Definition: A function $f$ may be specified by primitive recursion if its domain can be expressed as $S \times D$ where $S$ is ordered by $n:S \to N$, $N$ = the positive integers. Then $f$ is specified as $f(s,y)$ = if $n(s)>1$ then $h(s,y,f(m(s),y))$ else $g(y)$, where $m:S \to S$, $n(m(s)) = n(s)-1$ if $n(s)>1$, range $f$ = range $g$ = range $h$, domain $g = D$, and domain $h = S \times D \times$ range $f$. This may be applied with non-deterministic functions if the restriction on $m$ is obeyed, and for $f$ in a decomposition tree of an $f_i$ or an $f_{ij}$ with the proper ordering on component spaces and value spaces.

The operations of composition, subtree selection, and primitive recursion allow us to decompose the $f_i$ and $f_{ij}$ into lower level primitives. These primitives may express any functional relationship which we do not wish to further decompose. To express interaction we have three primitive exchange function types: XA, XC and XS. Their evaluations must synchronize with one another, so evaluation control and precedence must be discussed. Evaluating a state successor function $f$ causes each component successor $f_i$ to be evaluated once, which in turn causes value successors $f_{ij}$, if they were specified, to be evaluated some number of times, possibly zero. In function composition the functions $g_i$ must all be evaluated before the function $h$ is , in order to produce values for $h$'s arguments. In subtree selection some $g_i$ and $h_i$ may not be evaluated, and the order of evaluation of those evaluated is constrained by the if-then-else expression. In primitive recursion the function $h$ is evaluated some number of times, possibly zero, and the order of evaluation is given by the generated composition tree and the if-then-else expressions. These are the only constraints on evaluation without exchanges. However, any evaluation of an exchange must pair with the evaluation of some exchange with the same subscript, according to the following constraints. An $XC_i$ may pair with an $XC_i$, $XA_i$, or $XS_i$. An $XA_i$ may pair with an $XC_i$ or $XS_i$. An $XS_i$ may pair with an $XC_i$ or $XA_i$, except that an evaluation of an $XS_i$ may pair with itself. An evaluation of neither an $XC_i$ nor an $XA_i$ may pair with itself. Thus an evaluation of an $XC_i$ or an $XA_i$ must wait for the evaluation of another exchange with the same subscript and of the allowed type, but an $XS_i$ evaluation may pair with itself and evaluate immediately ($XS_i$ is called an immediate exchange). Every exchange has one argument; when evaluations of exchanges are paired, the output value of one exchange is the argument of the other. An $XS_i$ which pairs with itself thus evaluates to its own argument value.

Within the specification of a state successor function there may be several instances of exchanges with identical type and subscript, and with identical expressions for their arguments; these are all treated as if they were the same instance of the exchange. The set of them forms one half of a pair and all these instances will evaluate identically. If it is desired to distinguish between such identical instances, distinct second subscripts must be added to them. This will cause them to be evaluated separately.

## APPENDIX B – CONDITIONS ON EXCHANGES

Conditions on the use of exchanges which ensure algorithmic implication and boundedness of a system specification are given below.

Condition 1: In a specification by primitive recursion of a function f, with h and g as in the definition of primitive recursion in Appendix A, the specification of h may not contain exchanges. Furthermore the specification of a value successor function may not contain exchanges.

Without this condition the number of exchanges evaluated during the evaluation of a primitive recursively defined function or a value successor function could depend on state values--leading to an undecidable deadlock problem.

Definition: An exchange class is given by an index ($XC_i$ has index i, XCMESSAGE has index MESSAGE) and includes all exchanges in a specification which have that index.

Thus only exchanges in the same class are allowed to pair and exchange values with each other, so long as they meet the type constraints.

Condition 2: In a specification of a system of interacting processes, for any pair $(\Sigma,f)$ and $(\Sigma',f')$ of distinct processes, let I be the set of indices of those classes which have members in the specifications of both $(\Sigma,f)$ and $(\Sigma',f')$. Then the following must hold:

1) if any class $i \varepsilon I$ has a member in a third distinct process $(\Sigma'',f'')$, that member must have identical type and argument with a member in $(\Sigma,f)$ or $(\Sigma',f')$.

2) for each class $i \varepsilon I$ either

   a) one process has only members of type XS and the other process has only members of types XC and XA, or

   b) both processes have only members of type XC.

3) each process step (one evaluation of f or of f') entails either

a) evaluating exactly one member from each class $i \varepsilon I$, with no time precedence constraints between the different classes, or

b) evaluating no exchange from any class $i \varepsilon I$

4) both processes must execute a process step of type a) in 3) at bounded intervals.

Condition 2 is a sufficient constraint for algorithmic implication when a system complex has only two processes and there are no intra-process exchange interactions. If there are more than two processes we need:

Condition 3: In a specification of a system of interacting processes there must not be a set $(\Sigma_1,f_1)$, $(\Sigma_2,f_2)$, ...$(\Sigma_n,f_n)$ of processes and a set $C_1,C_2,\ldots,C_n,C_{n+1} = C_1$ of exchange classes such that, for each $i \varepsilon \{1,2,\ldots,n\}$, $C_i$ has a member in $(\Sigma_i,f_i)$ which is constrained to precede a member of $C_{i+1}$ in $(\Sigma_i,f_i)$, for any possible combination of choices in subtree selections.

Condition 3 is designed to prevent circular deadlocks. If there are no subtree selections then the precedence constraints in $(\Sigma_i,f_i)$ generated by composition and recursion (assuming Condition 1) are fixed. If there are subtree selections in $(\Sigma_i,f_i)$, then precedence constraints depend upon which functions are evaluated, and Condition 3 demands that the described deadlock loop may not occur for any combination of choices in subtree selection.

Conditions 1, 2, and 3 should be verified in that order; they ensure that a system without intra-process exchanges has algorithmic implication and boundedness.

Condition 2, part 3) ensures that no inter-process exchange class may be involved in intra-process exchanges.

Condition 4: *In a specification of a process* $(\Sigma,f)$, which may be part of a system complex, if any allowed order of evaluating $f$ results in two exchanges in $f$ being paired, then every possible evaluation of $f$ must pair those two exchanges.

Note Condition 4 disallows intra-process exchanges in subtree selections, and those involving XS type exchanges. Condition 4 states that the intra-process exchange pattern is fixed, and simplifies testing for algorithmic implication.

The formulation of conditions on exchange usage in specifications is at an early stage and is an important area of future work.

## APPENDIX C - VIRTUAL NETWORKS AND OPERATING SYSTEMS

The decomposition of a distributed data processing system into application systems, virtual operating systems, physical operating systems, and hardware systems seems essential to our developmental processes. A great deal of previous work on this problem has strengthened this belief. An introduction to some of this work is given in the draft manuscript included in this appendix. Further details may be found in [Kr 73] and [Co 75].

The work reported here was prior to the development of the present formal functional specifications and has not been reconsidered in this new context. It does serve to explore the virtual system issues, and so is included here.

[Kr 73] Kramer, John F., A General Structure for Uncooperative Processes Distributed over a System Network. Ph.D. Thesis, University of Wisconsin, Madison, 1973.

[Co 75] Cowan, George, Jr., Management of Resources in a Potentially Hostile Environment (Logical and Physical). Ph.D. Thesis, University of Wisconsin, Madison, 1975.

# THE ARCHITECTURE OF A MACHINE INDEPENDENT NETWORK OPERATING SYSTEM FOR HIERARCHIAL DELEGATION OF AUTHORITY

JOHN F. KRAMER, JR.
United States Navy

and

DONALD R. FITZWATER
University of Wisconsin, Madison

ABSTRACT: Networks of digital computer systems are in use today and there is little doubt that their numbers will continue to grow. If these endeavors are to avoid the traps that befell many of the third generation computer operating systems, a practical solution to the problems of system managability, application generality and process portability must be found. This article presents a set of postulated design constraints and a network design derived from them which is a practical solution to these problems. Through a clear factorization of the management responsibilities involved in a network, the design presented is able to permit both the implementers and the users of the network to optimize the management of their own resources with respect to their own criteria.

Key Words and Phrases: network, system, operating system, portability, processes, processor, cooperative processes, process communication.

This research was supported in part by the United States Navy under the Doctoral Study Program. Authors' addresses: John F. Kramer, Jr., 3-M Officer, Nimitz (CVAN 68) Precommissioning Unit, Newport News, Va. 23607; Donald R. Fitzwater, Department of Computer Sciences,

1210 West Dayton, University of Wisconsin, Madison, Wi  53706.

# OUTLINE

1. Introduction---management breakdown

    A. Centralized

    B. Decentralized

    C. Resource factorization

2. Postulated Design Constraints

    A. Network

    B. Responsibility

    C. Authority

    D. Delegation

    E. Modification

3. Systems Structures

    A. Level of Interpretation

    B. Digital Systems

    C. System Processes

    D. Network Processes

4. System Interactions

    A. Phase One

    B. Phase Two

    C. Phase Three

    D. Message Delivery

5. Network Processes

    A. Network Process State

1. Interface Buffer

2. Control Points

3. Expression States

4. Environment

5. Intra-process Transformation

B. Network System Language

1. GP

2. Roles

3. Type Conversions

4. Multi-Lingual

C. Process Tree

1. Conflict Resolution

2. Unique Authority

3. Birth and Death

D. Process Movement

1. Need For

2. Transmission

3. Reliability

4. Translation

6. Resource Management

A. Resources

B. Design Requirements

C. AO

D. Control of Interations

7. Operating Systems

   A. Implementation

      1. Goals

      2. Machine Dependent

      3. Language Effective

      4. Cooperation

   B. Network

      1. Goals

      2. Machine Independent

      3. Language for Application Independent of Machine

      4. Non-cooperation

      5. Authority---properly nested

   C. Common Subprocessor

8. User Freedoms

   A. Designate Process Control

   B. Standard Interface

   C. Complex Process Structures

   D. Control Point

   E. Modification and Evolution

## 1. Introduction

The emergence of computer networks has underscored the problems associated with the design of general purpose computer systems. Contemporary operating systems are prima facie evidence that in spite of nearly prohibitive investments, operating systems rarely remain in an error free state, rarely are exploited easily by users and rarely free the users from having to re-develop their application software to adapt a new computer. The enormous costs of systems today are caused in part by merely elaborating on the concept of centralized management of resources in a master/slave mode with little interaction between "slaves". This design concept was developed, and worked well, for early batch processing systems. In order to prevent chaotic conflicts between users due to hardware resource demands, operating system designers were forced to usurp most resource management decisions and become "masters".

If users are not allowed to manage resources to fit their diverse needs then the operating systems must attempt to do so for them. A proliferation of options and compromises emerge which rarely satisfy the knowledgeable user and significantly constrain applications. A network only enlarges the community of interacting users and points up the facility of continuing this ad hoc trend. It is not realistic to expect operating system designers to continue to make all of the management decisions in an efficient fashion while the diversity and complexity of user demands continues to grow.

In conventional systems, resources are usually managed by isolating each job or run and allowing it to interact solely with the operating system. Since only the processes of the operating system can directly interact, with sufficient care they can be managed to behave cooperatively with respect to resource demands. As soon as users are permitted to employ interacting processes using shared information, such cooperation can no longer be assumed or guaranteed. Processes may become uncooperative through malice, error, or lack of facilities for cooperation. Much of the capacity of systems today is spent managing user specified, unreliable processes. This is the "last straw" which causes conventional operating system designs to bog down. The work reported here is part of a detailed design study found in (8). This paper describes primarily the nature of the resulting network. Only suggestions of the justifications for the decisions made are presented here.

Decentralized management capabilities must be provided in a viable general purpose network so that each involved manager can exercise sufficient authority to meet that managers' goal. The partitioning of overall management responsibilities must be carefully done so as to prevent insoluable managerial conflicts. A "top down" design technique as used for this network design can guarantee such a partitioning with minimal constraints on the policies of each manager. We can clearly distinguish three kinds of management:

1) Implementation managers are responsible for the operation of a physical node in the network. This includes the support of virtual systems in a virtual network and the mapping of virtual resources onto physical resources via physical "operating system" processes. Aside from physical communication protocols, each

physical node may be designed, implemented, and managed independently. Management goals might be maximum investment returns or node utilization.

2) Network managers are responsible for the formal specification and evolution of the virtual network. As well as the initial virtual network operating processes. The virtual network serves as the interface between the implementation managers and the virtual process managers. Management goals include the maximal delegation of responsibility (and correspondingly factored authority) to the other managers.

3) Process managers are responsible for the specification and operations of the virtual processes via virtual system programs. Management goals include the creation of suitable virtual "operating system" processes as well as "application system" processes.

We are not here concerned with implementation management decisions, and will use unqualified "network", "system", and "process" to refer to their virtual counterparts.

The recognition of these three kinds of management is required for effective management decentralization. We also obtain a substantial amount of freedom to exploit changes in implementation technology while preserving our investment in application processes. As an additional benefit, our application processes can become portable over the network since they can be expressed in terms of (virtual) network programs. These programs may be interpreted by hardware, micro programs or normal programs, thus playing the role of machine independent, network "job control" programs. The compilation of subsets of the network system language to machine dependent "typed values" (with virtually inaccessible representations) will allow current jobs (e.g. Fortran programs) to run with normal efficiency, while allowing inter-process interactions over the entire network when desired. We are here primarily describing the resulting network design itself although much more could be said about implementation and application designs.

The network design itself is hierarchically structured with corresponding processor and process design decisions. For example, a processor component may require a particular process state component structure. Equally important, the process of network design is also hierarchically structured, in a "top down" manner. Each level in the design hierarchy has the goal of making only necessary (or non-controversial) design decisions so as to defer to later design levels all other decisions. This technique results in a cleanly structured network that delegates maximum freedom (through invariant network properties) for other managers to do their job.

## 2. Postulated Design Constraints

In view of the previous arguments, and others as detailed in (8), we postulate the following set of network design constraints to form a basis for further design decisions. Many other useful constraints may be derived from these but there is not space here to discuss them.

A. Network Communication

"The designed network must consist of a set of interacting, bounded, programmable, digital computer systems with well-defined processes, and be open to communication with foreign systems having undefined processes."

B. Responsibility factorization

"Responsibility for meeting the postulated design constraints and any design decisions must be factored between the network, implementation, and process designers."

C. Authority

"Each designer must have sufficient authority to define the effects, on the processes running in the network, of the interactions for which that designer has been delegated responsibility."

D. Delegation

"Although processes must be permitted to delegate to other processes their authority to control process interactions, the delegating process always retains the responsibility for that interaction and the authority to control the process to which it was delegated."

E. Modification

"The network definition must provide for modification of its definition. The design must provide sufficient constraints to be able to delegate safely to process managers all modification decisions and authority to control the effects of such decisions on the processes."

## 3. System Structures

In addition to not depending on the hardware/software of today, we must formally define our system structures so that each implementer of a node in the network can test the implementation (i.e. there is at least a standard, correct, specification). The logical structures which make up the network described here are formally defined in (8) using the technique defined in (6). This technique involves an effective network specification which allows a test of hypothesis about specified system behavior. For the purposes of this paper we will use informal, hopefully intuitive, notions of these system structures.

In general it is solely a matter of interpretation which determines whether a set of information physically represented in a computer system is data, program, processor, or system and is not an intrinsic property of the information itself. Since we must first choose a level and consistently describe structures from that level we must provide a few definitions. A natural choice for this paper is the network definition level since this serves as the partitioning interface between network users and implementation programmers. For the postulated design constraints presented above to be effective, a general model of digital systems and processes must be defined.

A digital system is composed of two parts, a system processor and a system state. A system processor is invariant to the processes which it interprets and contains a set of operators on the system state. These operators are applied by the system processer to define a new system state or to construct messages for transmission to other system states in a set of digital systems. The execution of a system processer occurs in two distinct phases. Given an initial system state, the processor evaluates all operators which apply and produces a new local system state. Each operator can be executed in parallel and without side effects on the other operators being executed. The local system state is then updated by any messages from other system processors and the local interpretation cycle begins again. This sequence is called a system step and transforms the system state into a successor state. This discrete sequence of states is called a digital computation. A set of digital computations represents a digital process. The interpretation cycle of each system in a set of interacting digital systems is independent and asynchronous. All inter-system communication will be defined in terms of asynchronous message transmission with any desired synchronization explicitly carried out by the cooperating digital processes. Messages will be received,after a finite delay,in the same order as they were transmitted by a given system. We will define some particular digital systems as network systems, and a set of such systems as a network. Note that such sets of digital systems could model ordinary computers at several levels of abstraction from one (quite abstracted) system to a network of "flip flop" and "gate" digital systems. There is a universal simulator of such sets of digital systems. Thus the system specification is already in a form that can be studied and tested. We do not require physical implementation constraints on how many networks or network systems may be implemented on a single physical system or on how many physical systems are used for a network or network system.

Our network system states are defined as a set of elements. Each element can be interpreted as the state of a synchronously interacting process component of the overall system process. All system state information and all system computations are based on these system process elements. We will constrain our network system operators so that at most one operator will modify a given system state element in a given interpretatic step. This not only prevents true race conditions from arising but also avoids the thorny question of how to merge several modifications of a given system process element into a well defined successor element With this design constraint we are free to define our operators as any total function of the corresponding state elements, producing a new state element.

Although we wish our system processors to be well-defined with respect to their operations on the system state, we desire also to permit them to communicate with systems outside of the formally defined networks which are not well-defined. Such systems are called foreign systems and are not constrained in their internal structure by the network designers except for communication conventions they must use if they wish to interact with a network systems. Such systems can be thought of as sources and sinks for messages. Certain elements of the system state are housekeeping processes that manage inter- and intra-system interactions. These elements contain system state information and are neither explicitly transformed by application programs nor transportable between systems in the network. Those elements of the state that are explicitly programmed, transformed, and moved from system to system will be called network (i.e. user) processes. A network process step consists of a cycle of three system process steps or phases. Two of the system phases are used for housekeeping functions and will be described later. Operators applied during phase two of a network process step will be called system subprocessors and their effect is local to that network process state. The complexity of a given network process step is therefore defined by the corresponding subprocessor. The process state defines the current address space of the process.

It is not feasible to present here the detailed arguments and justifications for this network design. We will refer the interested reader to (1) and subsequent reports in preparation. We will attempt to describe the major network system structures in the following section. Figure 1 is an informal characterization of one network system.

In review, a foreign system is not characterized (or constrained) except as a source or destination of messages. A network system is composed of a system processor and a system state. A system processor is composed of a set of operators, some of which are subprocessors. The system state is composed of system housekeeping elements and system elements interpreted as network (user) process states. A network consists of an inter-communicating set of such systems and foreign systems.

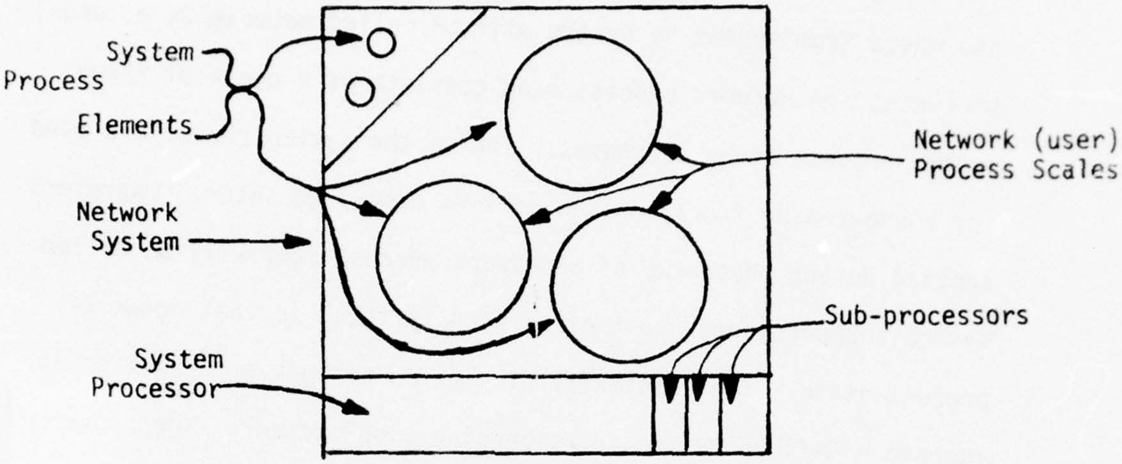


Figure 1: An informal characterization of a network system.

4. *Network System Interactions*

Inter-system interaction mechanisms are inherently non-local in nature, which means that the responsibility for their design falls to the network designers. The mechanism selected here is quite similar to the notion of a hierarchical interrupt system. In phase one, the system processor determines which sub-processors (if any) should be applied to each network process state in that system and delivers the appropriat messages (if any) to an interface buffer in each network process state. In phase two, the system processor applies the selected sub-processors to the selected network process states thus causing them to undergo a network process step. In phase three, the system processor performs the non-local services requested (if any) by messages left in the interface buffer by a sub-processor. Non-local operations are those which have side effects external to a network process state. Messages may be received by the system processor in all phases and are buffered until their delivery in some subsequent system phase one.

The system processor must be able to recognize, in any network process state, which sub-processor to apply in phase two and each sub-processor must be able to recognize its own state information component in that network process state. For these purposes, each network process state will contain one or more objects called control points. Although control points are used for a variety of purposes, only those aspects affecting sub-processor application will be described here. A control point roughly corresponds to an interrupt level in conventional systems.

All control points in a network process state are ordered by priority within that state. Each control point has an ordered set of channel terminations and buffers associated with it in the system state elements for the receipt of messages (not just an interrupt bit). Control point buffers may be armed or disarmed, and if disarmed, do not accept new messages. Control points may be enabled or disabled and active or inactive. When disabled, no further processing of pending messages in the control point buffers occurs. While active,a control point is a candidate to have a subprocessor allocated to it and further messages remain pending in the control point buffers. When a control is inactive but enabled and has a message pending, it is considered as a candidate for message delivery, conversion to active state and subsequent subprocessor application. The highest priority control point candidate present in a network process state, will be made active and the pending message will be placed in the process interface buffer. Two methods of message delivery are provided depending on a message handling status bit. Either the first message in the ordered control point buffers is delivered or an ordered concatenation of all messages for that control point is delivered. Message overwrite occurs if a message arrives at an armed buffer with a previously undelivered message in it. This may be avoided if desired by programming processes so they use appropriate synchronizing messages.

During phase two, the appropriate subprocessors are applied to network process states containing the active control points selected during phase one. At most one subprocessor is applied to a given process state during any phase two step. Although many process states in a given system may be requesting the same subprocessor, they each have it applied in parallel during phase two. Whether they are really serviced in sequence or in parallel is an implementation decision that affects only the duration of phase two since no inter-process interactions in this network system can occur until phase two has completed. All processes containing active control points will have sub-processors applied prior to the end of phase two.

Each sub-processor transforms only the network process state to which it is being applied. The applied sub-processor thus defines the successor network process state. If a system service is requested (a non-local transformation), the sub-processor will complete its network process step leaving a message, requesting the service, in the local interface tuffer for subsequent phase three processing. Note that network processes in the same system will proceed in synchronous parallel with each other, each completing one process step in each system cycle. Network processes in different systems will run asynchronously parallel.

During phase three all requests for non-local services which were left in the interface buffer are acted upon. There are four types of these services, message transmission, resource transmission, process state transmission, and system modification. Phase three ends when all such services have been completed and the interface buffers are cleared

If the interface buffer contains a request for message transmission, the associated message is removed from the buffer and "broadcast" to all systems but addressed to a particular control point buffer. The system containing the control point (destination) buffer will receive it, other systems will ignore the broadcast. Remember that a control point may have several destination buffers associated with it and that they are maintained as system state elements. Implementers, of course, are free to keep records of which system a control point resides in to permit them to transmit to only one system if they so desire. The relationship of these network system state elements is shown in Figure 2. The message has a network standard format and is represented as a string over a network standard character set.

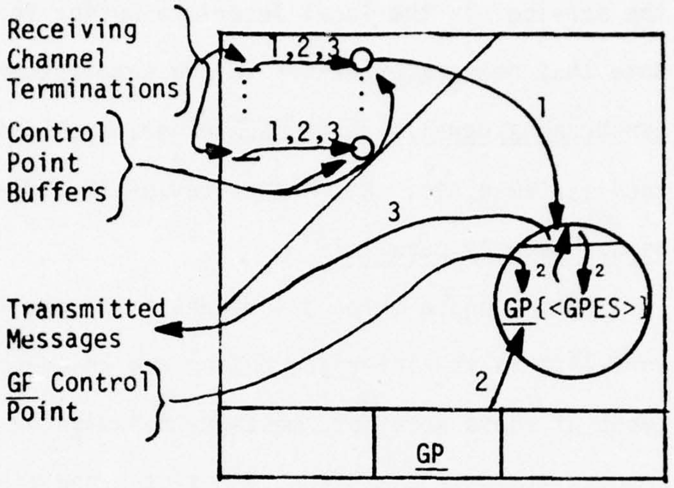


Figure 2: Intra-system message paths. The integers labeling communication transitions identify the system phase of that transition. GP is the sub-processor that interprets the network system language. <GPES> contains GP state information. The system step to transfer channel termination message to the corresponding control point buffers can be bypassed for messages arriving, at end of phase 3, if they are to be delivered on the next phase 1.

Although messages broadcast during phase three are subject to unspecified transmission delays (unless messages are intra-system), the order of transmission between any two systems is preserved in perception. In the receiving system, messages are placed in channel termination buffers and used to update control point buffers during every phase. A new message for a given buffer will overwrite the current contents (if any) of that buffer. If the destination control point is disarmed, the message will not be received at all. Figure 3 gives some examples of message interactions. In Figure 3 process A sends a message to process B in another system. B responds with a message for A. Process C sends a message to process D in the same system.

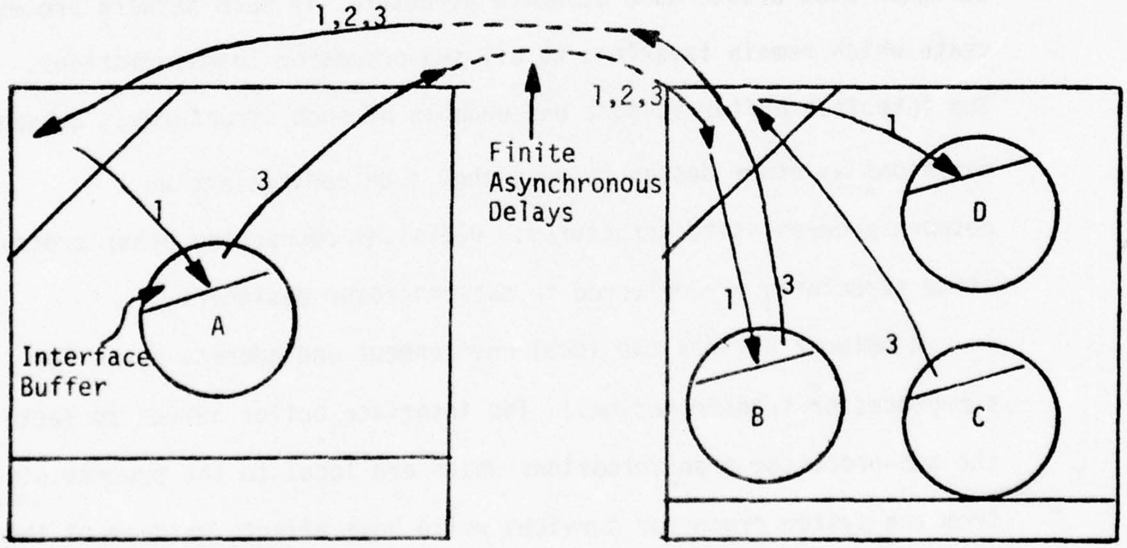


Figure 3: Interprocess Message Paths.
The Integer Labeling Communication Transitions
Identify The System Phase of that Transition.
The Letters Identify NEtwork Process States.

Note that the direct mechanism provided for interaction between processes involves message transmission from a network process state to some control point in a network process state. By controlling the access to destination control point names, network processes can be arbitrarily isolated so that uncooperative behavior effects can be localized. In addition, as far as network processes are concerned, message transmission is transparent to system boundaries and network processes may cooperatively move from system to system without losing communication or restricting their interactions.

5. Network Processes

Since non-local transformation services must be provided by the system processor rather than by subprocessors, the system processor designer must create some standard structures in each network process state which remain invariant to all sub-processor transformations. The interface buffer is just one example of such structures. Subsequent decisions by other designers must obey such constraints on network process state structures. Decisions concerning other process state structures are deferred to sub-processor designers.

A network defines the local environment and address space for sub-processor transformations. The interface buffer serves to factor the sub-processor transformations which are local to the process state from the system processor services which have effects outside of the process state. A process state will contain one or more control points, the status of which may be modified during system phase one as a result of message delivery or in system phase two as a result of the actions of an applied sub-processor. In addition to the role of control points in interprocess communication, control points also serve to delimit uniquely a portion of the network process state, called an expression state. Within this part of the process state the designers of the corresponding sub-processor are responsible for defining both the representations and the transformations which that sub-processor will carry out on those representations. All of the state information required for a sub-processor to continue a computation must be part of any corresponding expression state. A network process state thus contains an interface buffer, a set of expression states and one or more control points.

By constraining all virtual inter-process interactions to communications, and providing complete control over communications via restricted access to destination names, the network designers can supply sufficient mechanisms for application designers to isolate (to the desired extent) uncooperative computations. Having provided for worst case isolation, the network designers have an equal responsibility to provide for maximaly cooperative computations. With the restriction that at most one subprocessor will be applied to a given process state in system phase two, we can permit multiple expression states of multiple types to interact as desired by their subprocessor designers and even access and transform structures and values in other expression states in the same process. We must insist that access and transformation of a given type expression state by subprocessors of different type obey the rules established by that given type.

The critical section conflicts that can arise in intra-process transformations by competing expression states are not (and can not be) prevented by network designers who must instead, provide sufficient primitive operators so that any desired cooperative conflict resolution technique can be used. The network designers do not have to deal with such intra-process conflict, or guarantee that the conflict will be resolved, since the process will still be well defined and any side effects can be restricted to the process with the conflict. Multiple expression states in a network process state may be used as cooperatively as desired with minimal constraints from network designers.

Within an expression state we can distinguish between explicit "go to" executions and implicit "fault" conditions. Both forms change the "instruction counter" values of an expression state under rules specified by the corresponding sub-processor designer. Both forms have effects local to the containing expression state.

Similarly we can distinguish between explicit "move control point" executions and implicit "activate control point" (by internal command or external message receipt) conditions. Activations of multiple control points in a network process state represent a hierarchical interrupt system with the associated expression states serving as interrupt handlers. The movement of control points among expression states represent a scheduling operation such as for co-routines, tasks, simula "processes", etc. Both of these forms are local to the process state.

We can also distinguish explicit and implicit inter-process movements of control points as resources (controlled access objects) as discussed in a later section on resources. An application program thus has a wide range of structures to use as benefits the application, assigning potentially uncooperative computations to isolatable network processes (running either synchronously or asynchronously) and exploiting the advantages of cooperative computations by putting them in a single process. A network process is capable of supporting all computations that can be carried out on a conventional single processor, multiprogramming system.

There is a sub-processor, called GP, in every network system. The GP expression state can be programmed, in the network system language, to provide or request all network services. GP expression states thus can specify invariant computations despite movement of the containing network process. The network system language thus plays many roles such as the following:

a) The network job control language.

b) The network high level "machine" language.

c) The network implementation language for operating "systems":

d) The base language for definitional extension via source language macros or compilers.

e) The base language for evolutionary augmentation.

f) The system invariant computation language.

A given programmer may use GP only as a definitionally extendible job control language, while suppling Fortran programs to a "type conversion" compiler that produces physical node dependent programs just as he is accustomed to do. The compiler in such a case would produce an "execute only" type value whose internal structure is virtually inaccesible. The execution efficiency could thus be unchanged. However a vastly more general computational environment structure can be specified for new applications. There is no unique system language required by the network system design and this design will work with many language structures. A description of a network system language, aminol (8 1/2), is beyond the scope of this discussion. Indeed, aminol will allow the definition and introduction of new sub-processors with their corresponding expression states so that the network system processor itself can become multi-lingual.

As a result of this design, the freedom to create a highly structured and locally managed process, while minimizing the operating system interference which is normally required to control side effects on other processes, can be delegated to application programmers. Although a process has almost total control over the management of its own operations, some other process must be able to control interactions outside a process, even without the cooperation of the process itself, in order to resolve conflicts.

The problem of deciding when a given interaction is improper can only be resolved by another process aware of the interactions. The system can not resolve conflicting claims of one process with respect to the behavior of another process since it may require intimate knowledge of the specific applications. In order to guarantee resolution there must be a responsible authority who can control and manage interactions between the warring parties. Our system imposes a hierarchy of uniquely designated responsible authorities in the form of a network process tree, as in Figure 4, to make such an arbitration. All authority rests initially in the root of the tree. Although delegation to a child process is allowed, each root of a sub-tree remains responsible to its parent for the interactions of the processes in that sub-tree. An uncooperative root of a sub-tree will still be accountable to its parent process. One of the responsibilities of the network designer is to ensure that a cooperative process can restrict inter-process interactions of its sub-tree of processes.

The lines of responsibility in Figure 4 that define the process tree must have a basis in an ability of each process to control, and in worst case, isolate and kill, its sub-tree of processes. These control mechanisms are provided by management of "rights to..." as protected virtual resources using the facilities of the A O (accountable object) sub-processor as discussed below.

The process tree can both grow, by creating new "leafs", and shrink, by destroying "leafs". An existing process may move, as permitted by the parental authority, to any network system known to that parent. Thus both the process tree and its distribution over the network systems can change dynamically as a result of process computations. Since these are intrinsically non-local operations, the G P sub-processor can only request them. The responsibility

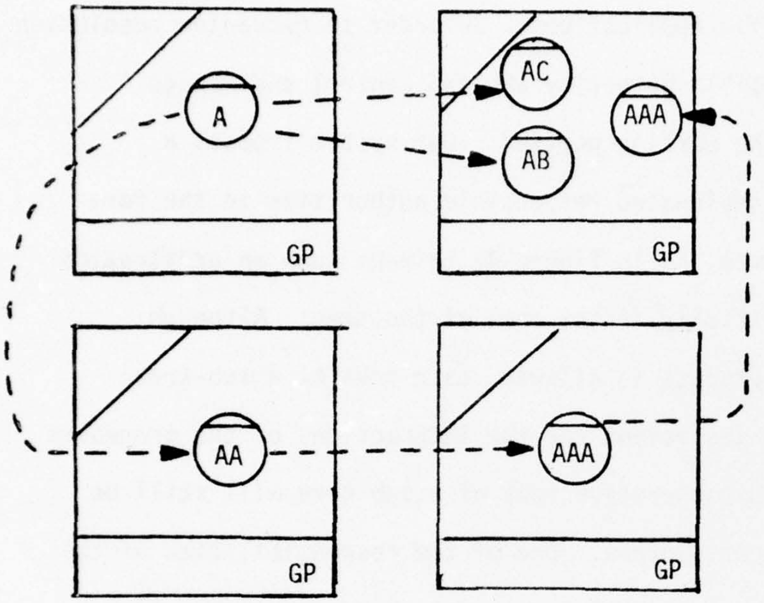
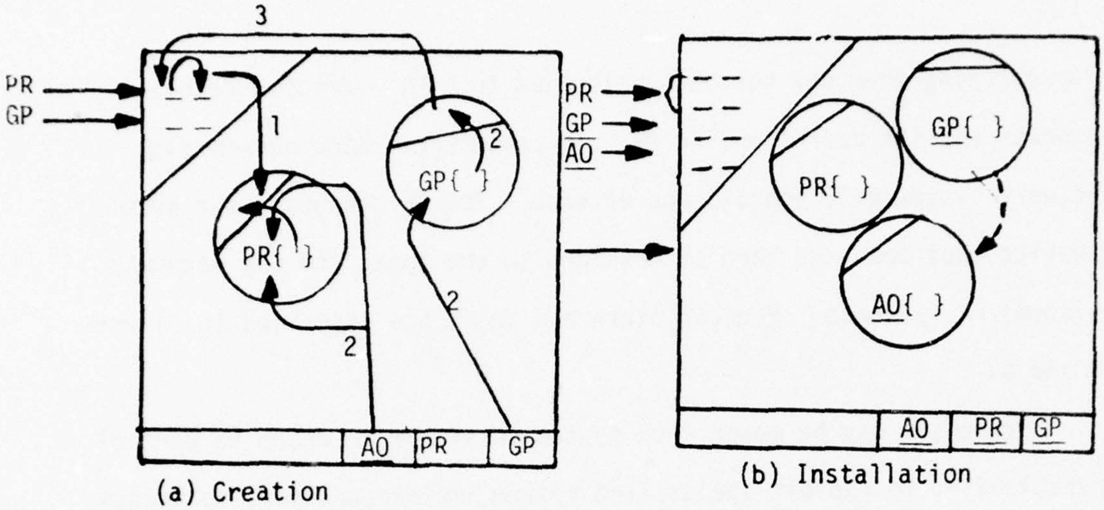


Figure 4: Network process tree lines of authority
Letters in process states identify the processes.
Conflict between processs "AAA" and "AB" can only be resolved by their least common ancestor, process "A".
By definition, no conflict can arise between process "AA", "AAA", and "AAAA" since each ancestor can arbitrarily control it's descendent.

for carrying them out has been delegated to a PR (Process Receive) process and its corresponding PR sub-processor. Each non-foreign network system will contain one of each. The GP requests for such service thus take the form of messages to the local (to the network system) PR process. Process birth and death are described in Figures 5 and 6.

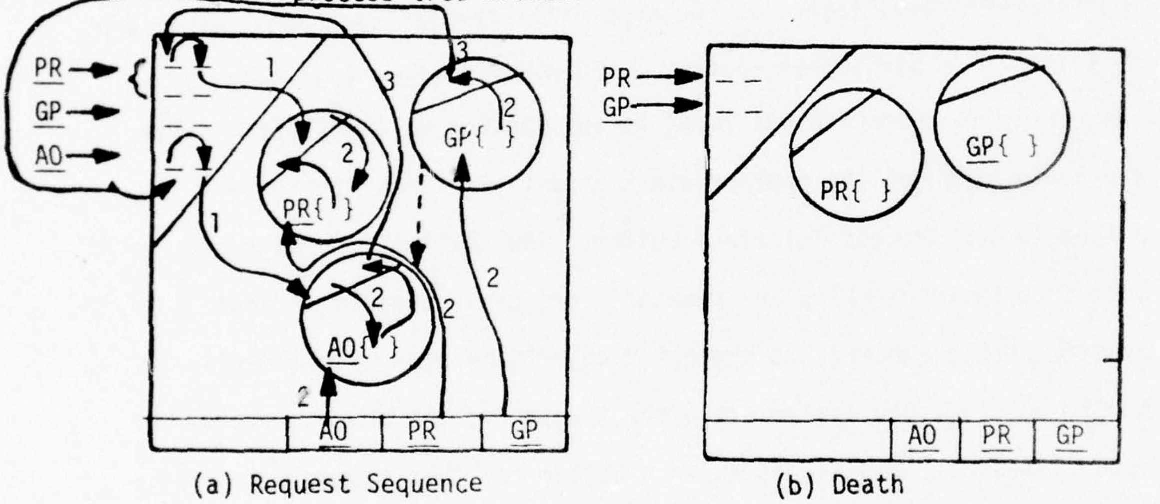
Processes may be moved from system to system in order to control parallelism, to exploit specialized system implementations, to access special sub-processors existing in particular systems, or to carry out inter-process communications local to one system instead of by inter-system communications. Accesses to a special data base may be more efficient in a particualr system. Such process state transmission is *inherently an operation not local to one process* or one system. When a process executes the appropriate transmit operator, the request is placed in the process interface buffer. The destination system is specified by referencing the name of a process contained in that system. Process names are protected objects managed as resources. During phase 3, the system processor transforms the process state into a "message" and places it in a buffer for that system's control point. The somewhat complex chores involved in transmitting and receiving process states are fulfilled cooperatively by the involved PR sub-processors. A unique PR process state containing only a PR control point and PR expression state is included in the system processes of each non-foreign system in the network. Since we can not guarantee the integrity of process states in foreign systems, we do not allow



(a) Creation

(b) Installation

Figure 5: Network process birth.
The integers associated with arrows identify system phases. The transformation between (a) and (b) occurs in the last system phase 3. Dashed arrow represents process tree branch.



(a) Request Sequence

(b) Death

Figure 6: Network process death.
The integers associated with arrows identify system phases. The "leaf" process is deleted in the last system phase 3. Dashed arrow represents process tree branch.

process states to be either transmitted to or received from foreign systems.

Communication of process states by the PR subprocessor must be reliable. We must permit an implementation to refuse to receive an additional process state without damaging the process in the transmitting system. This is done by not destroying the process state in the sending system until the receiving system acknowledges receipt of it. If a rejection response is received, the process state is "revived" in the source system and the original transmission operator in the revived process is faulted. Process management may then do as they like to ameliorate the problem.

The conversion of a process state to a message in the source system and the inverse operation in the destination system can easily be defined formally in the network. The corresponding operators in an implementation will of course be implementation dependent translators of the network defined form.

## 6. Resource Management

A process may cooperatively transmit any object to another process, but once the object is in the destination process state it is at the mercy of that process state. If the transmitting process wishes to restrict the access, transformation and disposal of the transmitted object we must provide facilities to guarantee the validity of the source imposed constraints even if the destination process is uncooperative. All control of inter-process interactions is based on the distribution of "rights to..." by the root of a sub-tree. We will call all such constrained objects resources in the virtual systems, and delegate the responsibility for enforcing those constraints to the AO (accountable object) sub-processor. The AO designers in turn, delegate all possible responsibility to process management defined programs, while remaining within the constraints placed on them by the network designers. There are three services that must remain with the AO sub-processor: that of reliable (guaranteed cooperative) communication, that of protecting access controls of a resource, and that of pre-emptive return of resources from uncooperative sub-trees. In order to ensure resource constraints, each network process state will contain a unique AO control point and AO expression state. Thus resources are passed to guaranteed cooperative AO expression states that enforce the constraints whether the process is cooperative or not. Thus the process tree "lines of responsibility" are embodied in cooperative AO communications and all resources are constrained to movements over the process tree. We thus prevent any conflicts of authority over a resource since, for each sub-tree, the root process is uniquely responsible. The detailed derivation and design of the AO sub-processor and its associated expression state is given in (3). We will only indicate the nature of the design here.

Unblockable communications are guaranteed by assigning the AO control point in each process state the highest priority in the process. Guaranteed asynchronous communication is provided by creating AO receiver buffers for the parent and for each of the children. In order to acknowledge message receipt, a set (at least one) of response buffers are provided in the transmitting process. The number of processes transmitting an acknowledgement to a particular process is restricted by requiring that any such transmission uniquely designate a particular response buffer. Using these features it is possible to define a communication protocal that will provide the required communication services for resource management.

The AO expression state contains a set of resources and each resource contains an object and an ordered set of access procedures. All access to the object by programs of the containing process must be made by the execution of the ordered set of procedures. A process may create a resource by specifying any object and any access procedure. AO conventions guarantee the protection of the resource without constraining process management's resource specification.

The access procedure may allow the allocation of the resource object (or a part of it) to a child process, while adding a new access procedure which must be executed to gain access to the previously specified access procedure. Thus additional, properly nested, access constraints may be added during alocation to a child process. Figure 7 presents an example of the resource relationship.

Process management is delegated responsibility for the normal management, allocation, transformation and return of resources. The GP sub-processor will include suitable primitive operators for this purpose. When cooperation breaks down, pre-emptive return via AO services is required. Since a resource may be fragmented and sub-allocated in pieces according to the rules and access mechanisms defined by the resource creator, AO only returns the pieces and lets the owner put them back together.

The transmission of unwanted or spurious messages is an explicit non-local process interaction which must be controllable. Any transmitting operator is required to have, as an operand, a resource containing a "right to transmit" to a particular control point buffer. The creation and allocation of such permits is under the control of process management using GP programs. This permits a parent to deny to a child the ability to interact explicitly with any other process except that parent or a sub-tree created by that child.

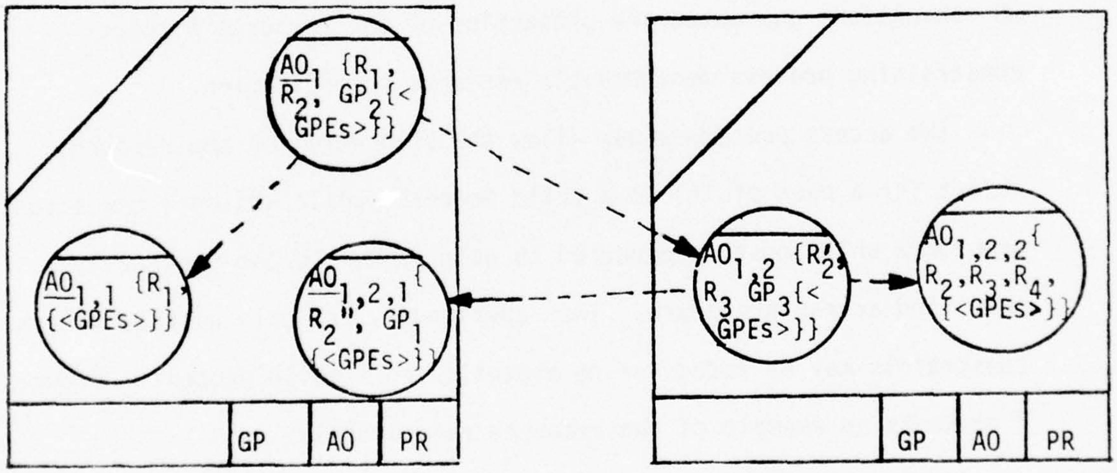

$AO_1$ {$R_1$, $R_2$, $GP_2${<GPEs>}}
$AO_{1,1}$ {$R_1$, {<GPEs>}}
$AO_{1,2,1}${$R_2''$, $GP_1${<GPEs>}}
$AO_{1,2}$ {$R_2'$, $R_3$ $GP_3${<GPEs>}}
$AO_{1,2,2}${$R_2$, $R_3$, $R_4$, {<GPEs>}}
GP
AO
PR
GP
AO
PR

Figure 7: Interprocess Resource Relationships

The tree structure is defined by the names (subscripts) of the AO control points. The AO expression state contains the set of resources and a GP expression state. A prime on a resource indicates an accessing program which has been added by other than the resource creator. Since each accessing program is additive, restrictions may be added but never removed by a child.

Implicit interaction between processes may occur as a result of conflicts for a particular set of implementation resources (physical). These interactions occur as the result of the exhaustion of some bounded phyiscal resource used in the representation of process states and must also be controllable by process management. Although we must not interfere with implementation management, we are entitled to delimit the scope of competition for such resources and define responsibility for its exhausion. By giving a child explicit delegation of a "limited competition permit" resource, a parent may restrict the competition of any child with that parent for such essential resources. When the limit of such a physical resource is reached, the effects are constrained to the sub-tree that was most locally restricted and will appear to the process as an "inaccessible object" in a process state. Inaccessible objects play a role similar to end of file marks on a tape drive and their detection by process management allows explicit programming for amelioration of the problem of recovery if desired. Competition restriction allows a parent freedom from a childs excesses in use of representational resources.

An example of the use of this restricted competition mechanism could be the partial control, in demand paging implementations, of replacement page selection.

## 7. Operating Systems

Network design responsibilities have been factored between the implementation and process managers. This requires a clean factoring of network resources (e.g. a file) and implementation resources (e.g. a Drum) upon which the logical network resources are maintained. Implementation designers clearly must be delegated responsibility for mapping network resources onto implementation resources. It is thus necessary to have two different operating systems, one for the virtual systems, and another for each of the physical node systems. The current size, complexity, and unmanageability of contemporary operating systems is in large part due to trying to meet these diverse goals with one operating system.

Physical operating systems may vary widely from node to node and may be developed independently subject only to minimum constraints placed on them by the network designers. Designers of each physical node can select some utilization function and then manage the physical resources in a way that optimizes this function. The implementation programming language used should provide easy exploitation of machine dependent resources and provide a highly optimized machine dependent implementation. In addition, at each node, programmers may be under one management and can produce cooperatively structured implementations by informal, but very hard to enforce, conventions. As long as the implementation can be debugged prior to its productive use in the network, inadvertent failures of cooperation can be prevented. Highly structured hierarchical layers of queue networks could be constructed as cooperative sequential processes and deadlock problems which can not be cooperatively prevented, can be cured by sacrificing jobs that can be re-run.

Like the implementers, the network managers (users) may have goals which vary markedly from process to process. The factorization above makes it possible for each of them to pursue these goals subject only to minimal constraints placed on them by network designers and network implementers. There is usually some utilization or reliability function which they are trying to optimize during their management of user created resources. They need a programming language that provides for the easy exploitation of machine independent resources and produces highly optimized (with respect to application measures), but machine independent, network processes as distinct from the implementers who are dealing with machine dependent resources and implementations. In the design presented here each process is subject to constraints imposed on it by its parent process. It can be held responsible for its actions while being free to manage its own affairs. It is possible to protect other network processes from interactions they do not wish to experience. Many users need to be able to develop programs simultaneously even though they are not all under the same management, and debugging operations must be able to proceed in the network while it is up and performing services for other processes. Inadvertent errors, as well as malicious non-cooperation by one process must not jeopardize system performance for others. When user program structures are based on cooperative hierarchies they often either become unpoliceable or they collapse under the required policing constraints. This independence means that deadlock problems can not be prevented or cured by reasonable algorithms imposed by the system. What is required instead are tools and facilities that allow delegation of such decisions to process managers who can choose appropriate tatics. In order to allow such application dependence network and implementation resources must be divorced. Although there still exists a "master/slave" mode, each process can now become the local operating system for itself and its children, subject only to the authority and constraints of its parent. This is advantageous since no global, premature, decisions need be made by a parent because now they can give their children the freedom to optimize their own operations while still retaining necessary control over them in case they go astray.

In order for process managers to meet their commitments they need a high level resource creation and manipulation language. In our network each system will provide a common subprocessor (GP) to interpret this virtual system language. Conventional and often unmanageable job control languages, with their many options and lack of general freedoms provide too rudimentary and specialized control of resources for processes engaged in defining operating environments for their children. A high level system language for network operating system implementation is required for this purpose.

Other application languages may be provided either by compilers producing output for the GP subprocessor or for other specialized subprocessors. To provide for such compilers, the system language interpreted by GP must, of course, be extensible. It must also be augmentable by implementation adaptions (possibly via micro-programming) if specific efficiency requirements for specific processes, as well as general evolutionary processes are to be permitted.

The design and specification of such a system language has been substantially completed and will be reported on subsequently. A prototype implementation of one complete network system will be completed in the near future. It should be pointed out that the overall network structure described here is not dependent on a specific GP design for its validity.

8. User Freedom

In the previous sections we briefly introduced some of the features of the network design. In this section we would like to present some of the freedom which these features provide the user. As can be seen from the design features presented, the network designers have not constrained in any way the design of user algorithms. The network is abstracted, both from any particular application of it and from any particular implementation of it. Different physical nodes supporting systems in the network, may use different physical assets to do so, with no concern for the users other than how it effects the costs of running their processes. The mechanisms invoked by a user to cause a subprocessor to be applied to a control point/expression state pair, or to use interprocess communication services are standard in each non-foreign network system. As a result, process states may be moved between systems using the services of the PR process and still be able to run in other systems without explicit changes to reflect the new system's implementation conventions.

Another important freedom provided by this design is the ability of a user to construct a single process environment containing multiple expression states, of different specialized subprocessor types, which may interact cooperatively. Very few constraints are placed on the subprocessors applied to such cooperating expression states by the network design other than the prevention of true race conditions (only one subprocessor at a time is applied to a process state) and the prevention of a violation of resource integrity (because of AO design). Adaptation of user specialized cooperative intra-process interactions is thus minimally constrained.

Control point communication provides several advantages to the user. First, since control points are paired with expression states, the destination of an interprocess communication can be bound to a particular expression state in a particular process. A sender need not be aware that a destination control point has been moved to a new process or that a whole process, containing the destination control point has been moved to another system. For the implementer, control point communication permits the maximum freedom to use any method of intersystem communication desired. The network could easily be implemented on the ARPANET (2, 4, 7, 9, 10) or on most of the other networks described in (1) and (5). Since the network design does not tell the implementer how to handle messages. The interrupt capability provides a significant advantage in terms of interprocess interactions. A control point/expression state can stop processing and go to sleep knowing that lower priority control points in that process will proceed until the higher priority control point either receives a message or is designated to run by a subprocessor operating on one of the other expression states in that process.

An advantage of using the system/subprocessor and process/control point/expression state concepts is that it becomes almost trivial to introduce new subprocessors into the network. The old processes still run the same while new processes can take advantage of the additions. The scope of the effects of such introductions will be limited to the processes introducing such subprocessors or to processes allocated the right to use them. Another advantage of providing a common interface is that it permits a user to move his process to other systems.

In addition to (8), the network design is more completely covered in (3) and other papers in preparation. It is of course not possible here to describe all of the aspects of this project. The development of this design was done with a cleanly factored set of design constraints which were abstracted without any particular technology or application in mind. This is one of the more important reasons why the design is a solution to many problems of manageability of processes in a network, of application generality, of process portability between systems in the network, and of network survival between changes in technology.

The design is fully implementable with the worst implementation being a complete simulation. An implementation currently exists of a single system, multiple process, and multiple expression state with multiple control points. The simulation of the functions of interprocess communication and control performed by the system processor is a relatively minor problem since the system processor implemented supports all but intersystem messages. It is not these functions which create implementation complexities. With the advent of microprogramming, overhead can be reduced, particularly in the implementation of subprocessors. As hardware, including firmware, becomes less expensive, the system processor functions of communications and subprocessor application can be easily implemented in them. Such implementations of these functions will be able to perform them in the range of several gate times if desired. The advantage of hard-wiring all subprocessors is something which should be further investigated. The choice between microprogrammed or hardwired subprocessors becomes one of balancing efficiency against flexibility.

Another important aspect of this design,not emphasized here, is that the network as designed has also been formally defined. It is important that the more formal properties of any design be investigated. A formal system provides us with an unambiguous, machine independent way of defining our results. Without such a system, the design might not be understandable by either the implementer or the user. Machine independent definitions are particularly necessary since we are interested in networks of systems and asynchronous computations. As a result of the formal definition system used, we get a well-defined concept of process and process step. The use of an interpreter for the definition system permits the design to be debugged. Analysis of the system definitions provides valuable insights into the process behaviors supported by the network.

One area introduced in (8) and to be pursued in subsequent reports is the possibility of permitting user control over certain modifications to the network design. This would include modifications of the formal definition to include new systems, new subprocessors, or even new operators in current subprocessors. One of the important constraints on such modifications is the verification that they will not improperly effect the processes in the network for which the user performing a modification is not responsible.

This brief introduction into the network design only touches on the basic structure of the network. The references can provide the interested reader with a detailed analysis of the design. We are interested in providing a network of interacting digital computer systems and structures to support general processes. This design provides user control over potentially uncooperative processes in a multiprocess computation. This includes the sharing of capabilities and permitting interactions of processes resident in different virtual systems. The design presented here has, we feel, accomplished these goals.

## References

1. Bell, C. Gordan "More Power by Networking," IEEE Spectrum, Feb. 1974, pp 40-45.

2. Carr, C. Stephen, Crocker, Stephen D. and Cerf, Vinton G. "HOST-HOST Communication Protocol in the ARPA Network," Proc. AFIPS SJCC, 1970, Vol. 36, pp 589-597.

3. Cowan, George Jr. Management of Resources in a Potentially Hostile Environment (Logical and Physical). Ph.D. Thesis, University of Wisconsin, Madison.

4. Crocker, Steven D., Heafner, John F., Metcalfe, Robert M., and Postel, Jonathan B. "Function-Oriented Protocols for the ARPA Computer Network," Proc. AFIPS SJCC, 1972, Vol. 40, pp 271-279.

5. Farber, David J. "Networks: An Introduction," Datamation, April 1972, pp 36-39.

6. Fitzwater, D. R. and Hintz, C. A. A System for the Formal Definition of Digital Systems. CS Tech Report #141, The University of Wisconsin Computer Sciences Department, 1971.

7. Heart, F. E., Kahn, R. E., Ornstein, S. M., Crowther, W. R. and Walden, D. C. "The Interface Message Processor for the ARPA Computer Network," Proc. AFIPS SJCC, 1970, Vol. 36, pp 551-567.

8. Kramer, John F. A General Structure for Uncooperative Processes Distributed Over a System Network. Ph.D. Thesis, University of Wisconsin, Madison, 1973.

9. Ornstein, S. M., Heart, F. E., Crowther, W. R., Rising, H. K., Russell, S. B. and Michel, A. "The Terminal IMP for the ARPA Computer Network," Proc. AFIPS SJCC, 1972, Vol. 40, pp 243-254.

10. Roberts, Lawrence G. and Wessler, Barry O. "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS SJCC, 1970, pp 543-549.

# APPENDIX D.

## D.1 Introduction

The following is an example of a functional specification using the notation developed in Section 3. The system specified is the network described in Section 5., and detailed in Appendix C. It should be noted that this system was originally specified using a different formal specification technique, thus the resulting design may not allow for the cleanest re-specification using the technique of Section 3.

Two specifications of the system will be presented here. The first is a high level specification whose functions are defined in terms of very high level primitives. The second specification is a more detailed version of the first. The form of presentation of both specifications is the same. First, the functions are defined with a short description accompanying each definition. Following this, the function definition tree and process graphs are illustrated. And finally the functions of the specification are summarized in a table. Preceding both specifications (and should be considered part of each) is a definition table of the value spaces and component spaces used in the specification.

## D.2 The Specifications

Both specifications consist of two state successor functions, 'SYS' and 'REAL WORLD'. SYS specifies a single network system and REALWORLD specifies the rest of the network as it appears to a single system.

Generally speaking, an application of SYS does the following:

1) Makes subprocessor and message buffer selection for each process state in the system.

2) Applies the subprocessors, leaving resulting messages in interface buffers.

3) Transmits messages.

The REALWORLD system contains a transmitter and a receiver and a packet of messages which have yet to be delivered for each system in the network. Roughly it operates as follows:

1) All receivers are fired in parallel picking up a packet (if any) sent from each system.

2) These packets are merged into one packet with an arbitrary choice made between messages for the same destination.

3) The resulting packet is distributed amongst the systems updating the packet containing all of the messages for that system which have yet to be delivered, new messages for this packet overwrite old messages with the same destination.

4) The updated packets are all transmitted in parallel from the REALWORLD system and any which do not get delivered form the undelivered packet for that system.

The reader is urged to refer to the function trees and process graphs when reading the specifications.

### D.2.1 Definition of Value Spaces and Component Spaces

VALUE SPACES

| | | |
|---|---|---|
| $V_p$ | ≡ | Space of process states |
| $V_{mB}$ | ≡ | Space of message buffers |
| $V_{IB}$ | ≡ | Space of interface buffers |

$V_{messages}$ ≡ Space of messages

$\{\$\}$ ≡ Space containing \$

$\{\epsilon\}$ ≡ Space containing $\{\varepsilon\}$

$V_{\overline{p}}$ ≡ $V_p \cup \{\$\}$

$V_{\overline{mB}}$ ≡ $V_{mB} \cup \{\$\} \cup \{\epsilon\}$

$V_{TO_i}$ ≡ Values of messages directed to system i

$V_{FROM_i}$ ≡ Values of messages sent from system i.

$V_{NEW}$ ≡ Values of messages in newly arrived packet (merged from all systems)

N ≡ Number of systems in network

$i \leq N$.

$t \equiv$ TRUE, $f \equiv$ FALSE

$\Sigma_p$ ≡ Component space of $V_p$

$\Sigma_{mB}$ ≡ Component space of $V_{mB}$

$\Sigma_{IB}$ ≡ Component space of $V_{IB}$

$\Sigma_{\overline{p}}$ ≡ Component space of $V_{\overline{p}}$

$\Sigma_{\overline{mB}}$ ≡ Component space of $V_{\overline{mB}}$

$\Sigma_{SYS}$ ≡ $\Sigma_p \times \Sigma_{mB}$

$\Sigma_{TO_i}$ ≡ Component space of $V_{TO_i}$

$\Sigma_{FROM_i}$ ≡ Component space of $V_{FROM_i}$

$\Sigma_{NEW}$ ≡ Component space of $V_{NEW}$

$\Sigma_{REAL}$ ≡ $\prod_{i=1}^{N} \Sigma_{TO_i}$

(Note: $\prod_{i=1}^{N} \Sigma_i \equiv \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n$)

## D.2.2 High Level Specification

The function tree and process graph for SYS appear in Figures 1 and 2 and for REALWORLD in Figures 3 and 4.

### D.2.2.1 High Level Specification of SYS

1) SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$

   $SYS(\sigma_{SYS}) \equiv SYS_1(\sigma_p, \sigma_{mB})$

   Comment: SYS is composed of a single component function.

2) $SYS_2$: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$

   $SYS_1 (\sigma_p, \sigma_{mB}) \equiv$ Phase 3 (SP(Status($\sigma_p, \sigma_{mB}$)))

3) STATUS: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$

   STATUS $(\sigma_p, \sigma_{mB}) \equiv$ PRIMITIVE

In general, the Status primitive will return pairs of (process state, message buffer) such that the process state is requesting that message buffer and the buffer is a buffer for the highest priority control point in the process state. If a message buffer is not requested a dummy is returned: (\$, message buffer). If a process is active and has (or wishes) no messages: (process state, $\varepsilon$) is returned, and if a process state is not to have a subprocessor applied it returns (process state, \$). Thus Status does the selection of the appropriate control point expression state in a given user state for subprocessor allocation, and associates the appropriate message with the process state. It returns a dummy pair for unused messages and also for unused process states.

4) $SP_i : \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$

$SP_1(\sigma_{\overline{p}}, \sigma_{\overline{mB}}) = SP_{11}(V_{\overline{p}}, V_{\overline{mB}})$

5) $SP_{11} : V_{\overline{p}} \times V_{\overline{mB}} \rightarrow V_p \times V_{mB} \times \Sigma_{IB}$

$SP_{11}(V_{\overline{p}}, V_{\overline{mB}}) \equiv$ PRIMITIVE

$SP_{11}$ applies the appropriate subprocessors to the elements of $(\sigma_{\overline{p}}, \sigma_{\overline{mB}})$, delivering the messages (if any) and leaving messages in the interface buffer. On dummy pairs $SP_{11}$ simply carries forward the process state in (process state, \$) pairs and the message buffer in (\$, message buffer) pairs. The value function models the parallelism of subprocessor allocation.

6) Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$

Phase $3(\sigma_p, \sigma_{mB}, \sigma_{IB}) \equiv$ PRIMITIVE

Phase 3 finishes up the process step by:

1) Transmitting the messages to external systems

2) Receive messages from external systems

3) Merge received messages with old message buffers

4) Merge message buffers with intra-system messages.

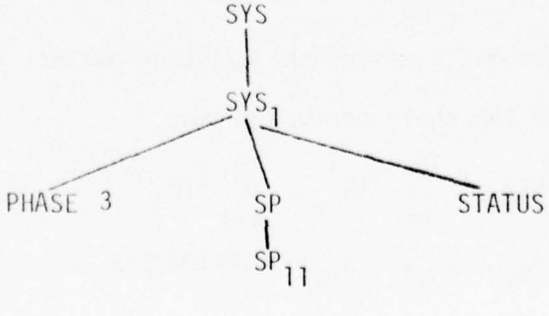


FIG. 1: Function tree for high level specification of SYS

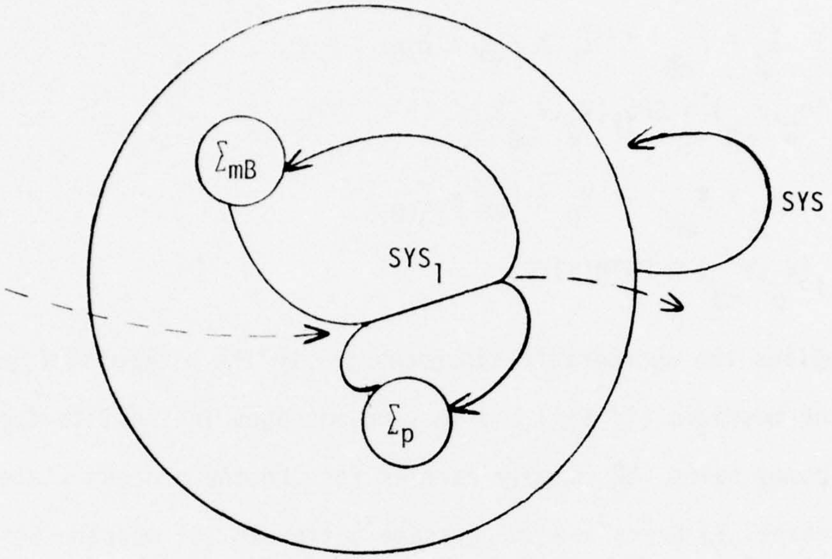


FIG. 2: Process graph for SYS

### D.2.2.2 High Level Specification of Realworld

1) Realworld: $\Sigma_{REAL} \to \Sigma_{REAL}$

   $\text{Realworld}(\sigma_{REAL}) \equiv \text{Realworld}_1(\sigma_{TO_1}, \ldots \sigma_{TO_N})$

2) $\text{Realworld}_1 : \prod_{i=1}^{N} \Sigma_{TO_i} \to \prod_{i=1}^{N} \Sigma_{TO_i}$

   $\text{Realworld}_1\ (\sigma_{TO_1}, \ldots, \sigma_{TO_N}) \equiv \text{Join}\ \ (\text{Packetin}(\ ), \text{Trysend}\ (\sigma_{TO_1}, \ldots, \sigma_{TO_N}))$

Comment: Realworld$_1$, merges the existing packets for each system with the newly arrived packet.

3) Join: $\Sigma_{NEW} \times \prod_{i=1}^{N} \Sigma_{TO_i} \to \prod_{i=1}^{N} \Sigma_{TO_i}$

   Join: $(\sigma_{NEW}, \sigma_{TO_1}, \ldots \sigma_{TO_N}) \equiv \text{PRIMITIVE}$

Comment: Join updates each $\sigma_{TO_i}$ with the messages in $\sigma_{NEW}$ for the ith system, overwriting old messages for the same destination.

4) Packetin: $\phi \to \Sigma_{NEW}$

Packetin( ) $\equiv$ PRIMITIVE

Comment: Packetin receives a packet from each system (if one was sent) and coalesces all received packets into one packet, choosing at most one message for each destination.

5) TRYSEND: $\prod_{i=1}^{N} \Sigma_{TO_i} \to \prod_{i=1}^{N} \Sigma_{TO_i}$

TRYSEND($\sigma_{TO_1},\ldots,\sigma_{TO_N}$) $\equiv$ PRIMITIVE

Comment: TRYSEND tries to send the accumulate packet for each system to that system. All packets are sent in parallel.

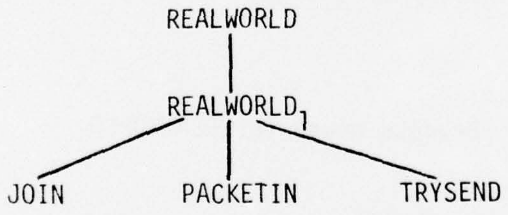


FIG. 3: Function tree for high level specification of REALWORLD.

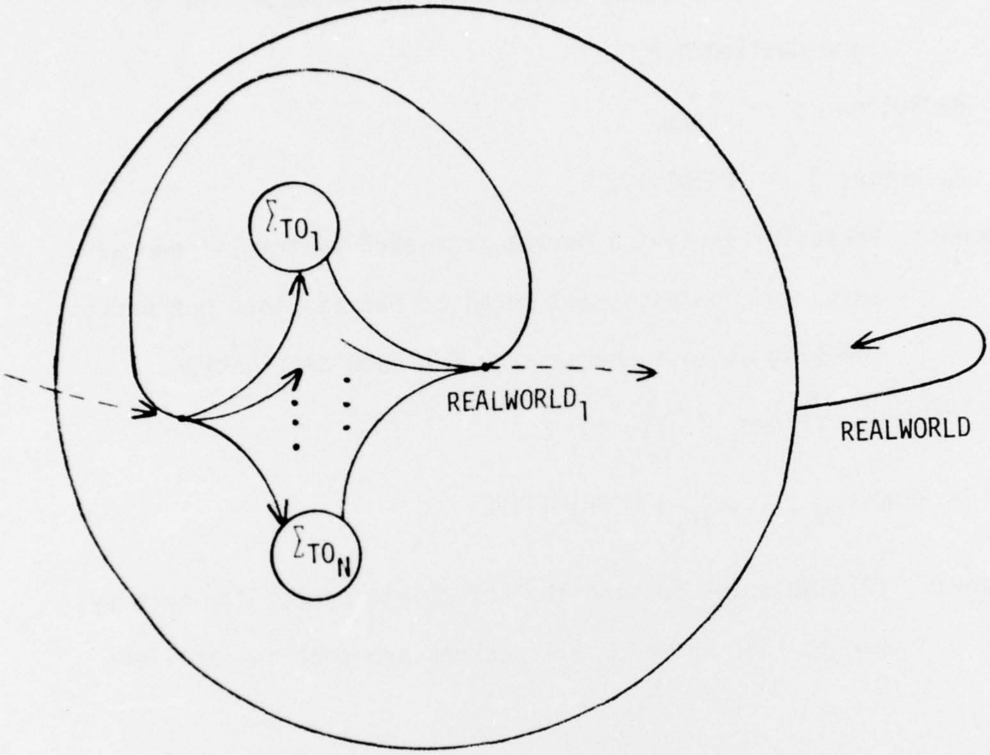


FIG. 4: Process graph for REALWORLD.

D.2.2.3 Function Table for High Level Specification

| TYPE | MAPPING | DEFINITION |
|---|---|---|
| State Successor | SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$ | $SYS(\sigma_{SYS}) \equiv SYS_1(\sigma_p,\sigma_{mB})$ |
| Component | $SYS_1$: $\Sigma_p \times \Sigma_{mB}\ \Sigma_p \times \Sigma_{mB}$ | $SYS_1(\sigma_p,\sigma_{MB}) \equiv$ Phase 3 $(SP(STATUS(\sigma_p,\sigma_{mB})))$ |
| Component | $STATUS_1$: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}}$ | PRIMITIVE |
| Component | $SP_1$: $\Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$ | $SP_1(\sigma_{\overline{p}},\sigma_{\overline{mB}}) \equiv SP_{11}(V_{\overline{p}},V_{\overline{mB}})$ |
| Value | $SP_{11}$: $V_{\overline{p}} \times V_{\overline{mB}} \rightarrow V_p \times V_{mB} \times V_{IB}$ | PRIMITIVE |
| Component | Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$ | PRIMITIVE |
| State Successor | Realworld: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$ | Realworld $(\sigma_{REAL}) \equiv Realworld_1(\sigma_{TO_1},\ldots,\sigma_{TO_N})$ |
| Component | $Realworld_1$: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$ | $Realworld_1(\sigma_{TO_1},\ldots,\sigma_{TO_N}) \equiv JOIN(PACKETIN(\ ),$ $TRYSEND(\sigma_{TO},\ldots\sigma_{TO_N}))$ |
| Component | JOIN: $\Sigma_{NEW} \times \prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$ | PRIMITIVE |
| Component | PACKETIN: $\phi \rightarrow \Sigma_{NEW}$ | PRIMITIVE |
| Component | TRYSEND: $\prod_{i=1}^{N} \Sigma_{TO_i} \rightarrow \prod_{i=1}^{N} \Sigma_{TO_i}$ | PRIMITIVE |

### D.2.3 Detailed Specification

The function tree and process graph for SYS appear in Figures 5 and 6. A process graph for SP appears in Figure 7 and the function tree and process graph for REALWORLD appear in Figures 8 and 9.

#### D.2.3.1 Detailed Specification of SYS

1) SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$

   $SYS(\sigma_{SYS}) \equiv SYS_1\ (\sigma_p, \sigma_{mB})$

   Comment: SYS is composed of a single component successor function $SYS_1$.

2) $SYS_1$: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$

   $SYS_1(\sigma_p, \sigma_{mB}) \equiv$ Phase 3 $(SP_1(STATUS\ (\sigma_p, \sigma_{mB}))$

   Comment: $SYS_1$ is defined in terms three component successor functions Phase 3, $SP_1$ and $STATUS_1$.

3) $STATUS_1$: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_{\overline{p}} \quad \Sigma_{\overline{mB}}$

   $STATUS_1(\sigma_p, \sigma_{mB}) \equiv$ PRIMITIVE

   Comment: The function of $STATUS_1$ is to pair up a particular process state with the appropriate message buffer. There are four possibilities:

   1) State $V_p$ is requesting $V_{mB}$ and $V_{mB}$ is a buffer for the highest priority control point in $V_p$.

   2) State $V_p$ is active and not requesting any messages, therefore $STATUS_1$ returns $(V_p, \varepsilon)$.

3) State $V_p$ is neither active nor has a message pending, thus $STATUS_1$ returns $(V_p, \$)$

4) Message $V_{mB}$ is requested by no $V_p$, $STATUS_1$ returns $(\$, V_{mB})$

4) $SP_1: \Sigma_{\overline{p}} \times \Sigma_{\overline{mB}} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$

Comment: In effect $SP_1$ will apply the correct subprocessor (if any) to the process states and messages (if any) resulting in new process states, message buffers and interface buffer values. To model the parallelism of what is effectively Phase 2 of the system step we will specify $SP_1$ in terms of four value functions.

$$SP_1\ (\sigma_{\overline{p}}, \sigma_{\overline{mB}}) \equiv \begin{cases} SP_{11}: & V_p \times V_{mB} \rightarrow V_p \times V_{IB} \\ SP_{12}: & V_p \times \{\epsilon\} \rightarrow V_p \times V_{IB} \\ SP_{13}: & \{\$\} \times V_{mB} \rightarrow V_{mB} \\ SP_{14}: & V_p \times \{\$\} \rightarrow V_p \end{cases}$$

(5a) $SP_{11}: V_p \times V_{mB} \rightarrow V_p \times V_{IB}$

$SP_{11}(V_p, V_{mB}) \equiv$ PRIMITIVE,

Comment: Delivers message $V_{mB}$, and applies appropriate subprocessor resulting in a new state and possibly a message in the interface buffer.

(5b) $SP_{12}: V_p \times \epsilon \rightarrow V_p \times V_{IB}$

$SP_{12}(V_p, \epsilon) \equiv$ PRIMITIVE, same as $SP_{11}$ except no message delivered.

(5c) $SP_{13}$: $\$ \times V_{mB} \rightarrow V_{mB}$

$SP_{13}(\$,V_{mB}) \equiv (t:V_{mB})$, carries forward unused messages

(5d) $SP_{14}$: $V_p \times \$ \rightarrow V_p$

$SP_{14}(V_p,\$) \equiv (t:V_p)$, carries forward unchanged process states.

Comment: See process graph for $SP_1$, Figure 3.

The subprocessors have been applied so it remains to specify the message transmission and receival.

(6) Phase 3 : $\Sigma_p \times \Sigma_{mB} \times \Sigma_B \rightarrow \Sigma_{mB}$

Phase $3(\sigma_p,\sigma_{mB},\sigma_{IB}) \equiv (t:(\sigma_p,\text{UPDATE}(\text{NEWIN}(\sigma_{mB}),\text{NEWOUT}(\sigma_{IB}))))$

Comment: Phase 3 merely carries forward the process states $\sigma_p$, and updates the message buffers.

(7) UPDATE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$

$\text{UPDATE}(\sigma_{mB},\sigma_{messages}) \equiv$ PRIMITIVE

Comment: Update merges the messages in $\sigma_{mB}$ and $\sigma_{messages}$ with the provision that if they each have a message for the same destination only the message in $\sigma_{messages}$ is kept.

In terms of the network system, Update models the fact that intra-system communications take precedence over inter-system communication.

(8) NEWIN: $\Sigma_{mB} \rightarrow \Sigma_{mB}$

$\text{NEWIN}(\sigma_{mB}) \equiv$ Compose $(\sigma_{mB},\text{REC}\{\ \})$

Comment: The value of NEWIN is the current message buffers merged with the incoming external messages.

(9) COMPOSE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$

COMPOSE($\sigma_{mB}$,$\sigma_{messages}$) $\equiv$ PRIMITIVE

Comment: Compose merges the message of $\sigma_{mB}$ and $\sigma_{messages}$ with $\sigma_{messages}$ taking precedence.

(10) REC: $\phi \rightarrow \Sigma_{messages}$

REC({ }) $\equiv$ XCLOCREC$_i${ }

Comment: The i indicates that we are in network system i., XCLOCREC$_i$ receives the external messages.

(11) NEWOUT: $\Sigma_{IB} \rightarrow \Sigma_{messages}$

NEWOUT($\sigma_{IB}$) $\equiv$ (t:(CONCAT(INSYS($\sigma_{IB}$)),SEND(OUTSYS($\sigma_{IB}$))))

Comment: NEWOUT sends the inter-system messages and carries forward the intra-system messages with messages for the same destination concatonated.

(12) CONCAT: $\Sigma_{IB} \rightarrow \Sigma_{IB}$

CONCAT($\sigma_{IB}$) $\equiv$ PRIMITIVE, concatonates all messages for the same destination into one message and carries forward the others.

(13) INSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$

INSYS($\sigma_{IB}$) $\equiv$ INSYS$_{11}$($V_{IB}$)

Comment: INSYS is specified as a value function.

(13a) INSYS$_{11}$: $V_{IB} \rightarrow V_{IB}$

INSYS$_{11}$($V_{IB}$) $\equiv$ PRIMITIVE,

Comment: Value of INSYS$_{11}$($V_{IB}$) = $V_{IB}$ if $V_{IB}$ is to go to a destination in the system and $\phi$ otherwise.

(14) OUTSYS: $\Sigma_{IB} \rightarrow \Sigma_{IB}$

OUTSYS$(\sigma_{IB}) \equiv$ OUTSYS$_{11}(V_{IB})$

(14a) OUTSYS$_{11}$: $V_{IB} \rightarrow V_{IB}$

OUTSYS$_{11}(V_{IB}) \equiv$ PRIMITIVE, value is $V_{IB}$ if $V_{IB}$ is to go to a destination outside the system, and $\phi$ otherwise.

(15) SEND: $\Sigma_{IB} \rightarrow \phi$

SEND$(\sigma_{IB}) \equiv$ XCLOCSEND$_i(\sigma_{IB})$

Comment: SEND transmits the external messages, the i indicates that this is the ith system.

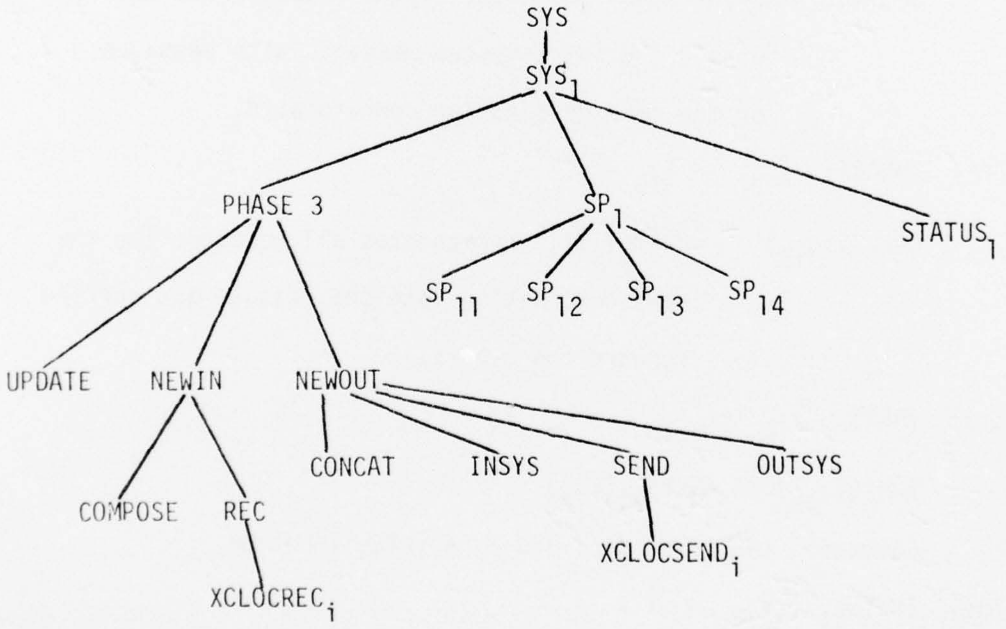


FIG. 5 Function tree for detailed specification of SYS.

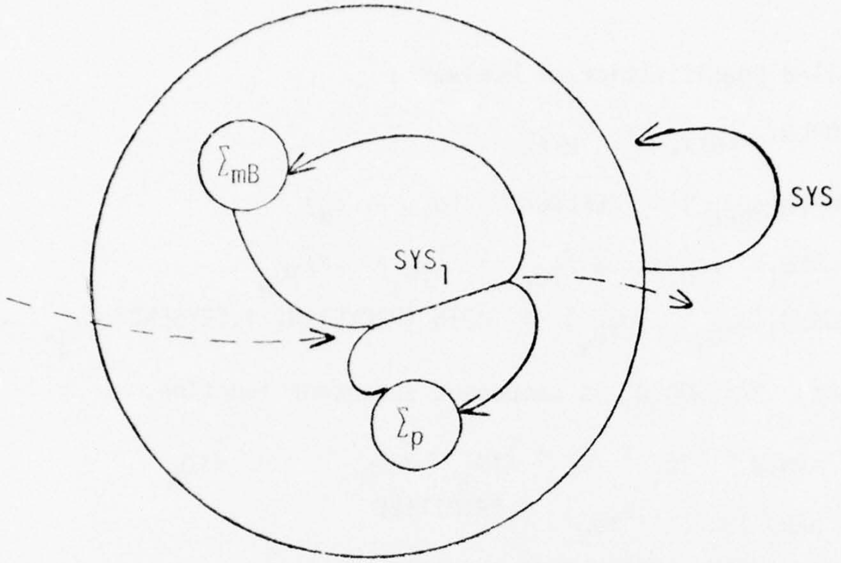


FIG. 6: Process graph for SYS

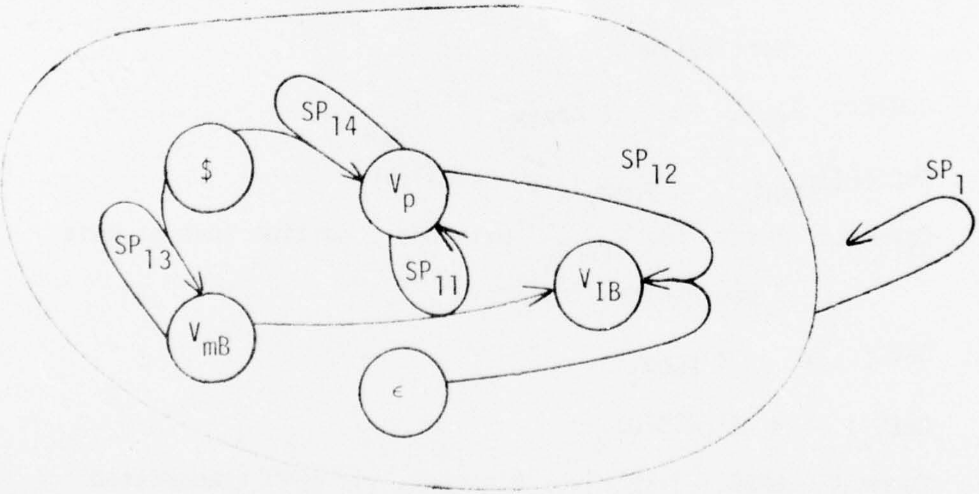


FIG. 7: Process graph for detailed specification of $SP_1$.

D.2.3.2 Detailed Specification of Realworld

1) REALWORLD: $\Sigma_{REAL} \rightarrow \Sigma_{REAL}$

REALWORLD($\sigma_{REAL}$) $\equiv$ REALWORLD$_1$ ($\sigma_1, \ldots, \sigma_N$)

2) REALWORLD$_1$: $\Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N}$

REALWORLD$_1$($\sigma_{TO_1} \ldots, \sigma_{TO_N}$) $\equiv$ JOIN (PACKETIN( ),TRYSEND($\sigma_{TO_1}, \ldots \sigma_{TO_N}$))

Comment: REALWORLD$_1$ is component successor function.

3) JOIN: $\Sigma_{NEW} \times \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \ldots \times \Sigma_{TO_N}$

JOIN($\sigma_{NEW}, \sigma_{TO_1}, \ldots, \sigma_{TO_N}$) $\equiv$ PRIMITIVE

Comment: JOIN updates the $\sigma_i$ from $\sigma_{NEW}$

4) PACKETIN: $\phi \rightarrow \Sigma_{NEW}$

PACKETIN( ) $\equiv$ CHOICE(GREC,{ },...,GREC$_N${ })

Comment: PACKETIN receives messages transmitted from all systems and merges them into one packet with at most 1 message/destination.

5) CHOICE: $\Sigma_{FROM_1} \times \ldots \times \Sigma_{FROM_N} \rightarrow \Sigma_{NEW}$

CHOICE($\sigma_{FROM_1}, \ldots, \sigma_{FROM_N}$) $\equiv$ PRIMITIVE

Comment: Merge the $\sigma_{FROM_i}$ with the condition that at most 1 message/destination.

6) GREC$_i$: $\phi \rightarrow \Sigma_{FROM_i}$

GREC$_i${ } $\equiv$ XSLOCSEND$_i${ }

Comment: GREC$_i${ } receives a packet, (if any) transmitted from the ith system.

7) TRYSEND: $\Sigma_{TO_i} \times \dots \times \Sigma_{TO_N} \rightarrow \Sigma_{TO_1} \times \dots \times \Sigma_{TO_N}$

$\text{TRYSEND}(\sigma_{TO_1},\dots,\sigma_{TO_N}) \equiv (t:(\text{GSEND},(\sigma_{TO_1}),\dots\text{GSEND}_N(\sigma_{TO_N})))$

Comment: TRYSEND tries to send the as yet undelivered packet for each system, to that system.

8) $\text{GSEND}_i$: $\Sigma_{TO_i} \rightarrow \Sigma_{TO_i}$

$\text{GSEND}_i(\sigma_{TO_i}) \equiv \text{XSLOCREC}_i\{\sigma_{TO_i}\}$

Comment: $\text{GSEND}_i$ attempts to send to the ith system.

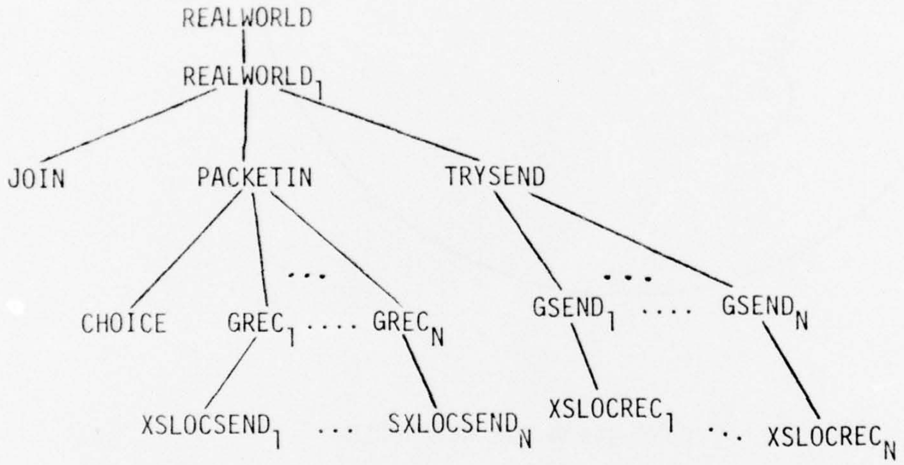


FIG. 8: Function definition trees for detailed specification of REALWORLD.

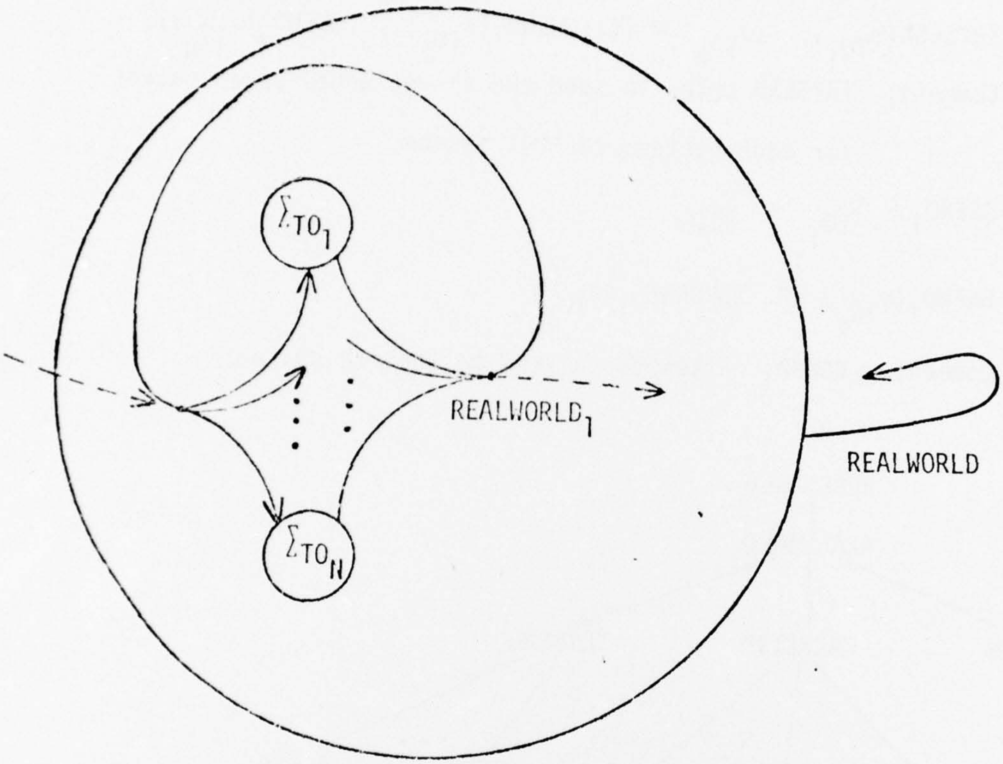


FIG. 9: Process graph for REALWORLD.

### D.2.3.3 Function Table for Detailed Specification

| TYPE | MAPPING | DEFINITION |
|---|---|---|
| State Successor | SYS: $\Sigma_{SYS} \rightarrow \Sigma_{SYS}$ | SYS $\equiv$ $SYS_1$ |
| Component | $SYS_1$: $\Sigma_p \times \Sigma_{mB} \rightarrow \Sigma_p \times \Sigma_{mB}$ | $SYS_1(\sigma_p,\sigma_{mB}) \equiv PHASE3(SP_1(STATUS_1(\sigma_p,\sigma_{mB})))$ |
| Component | $STATUS_1$: $\Sigma_p \times \Sigma_{mB} \rightarrow \underline{\Sigma}_p \times \underline{\Sigma}_{mB}$ | PRIMITIVE |
| Component | $SP_1$: $\underline{\Sigma}_p \times \underline{\Sigma}_{mB} \rightarrow \Sigma_p \times \Sigma_{mB} \times \Sigma_{IB}$ | $SP_1 \equiv SP_{11}, SP_{12}, SP_{13}, SP_{14}$ |
| Value | $SP_{11}$: $V_p \times V_{mB} \rightarrow V_p \times V_{IB}$ | PRIMITIVE |
| Value | $SP_{12}$: $V_p \times \{\epsilon\} \rightarrow V_p \times V_{IB}$ | PRIMITIVE |
| Value | $SP_{13}$: $\{\epsilon\} \times V_{mB} \rightarrow V_p$ | $SP_{13}(\$,V_{mB}) \equiv (t:(V_{mB}))$ |
| Value | $SP_{14}$: $V_p \times \{\epsilon\} \rightarrow V_p$ | $SP_{14}(V_p,\$) \equiv (t:(V_{mB}))$ |
| Component | Phase 3: $\Sigma_p \times \Sigma_{mB} \times \Sigma_{IB} \rightarrow \Sigma_p \times \Sigma_{mB}$ | PHASE $3(\sigma_p,\sigma_m,\sigma_{IB}) \equiv (t:(\sigma_p, UPDATE(NEWIN(\sigma_{mB}), NEWOUT(\sigma_{IB})))$ |
| Component | UPDATE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$ | PRIMITIVE |
| Component | NEWIN: $\Sigma_{mB} \rightarrow \Sigma_{mB}$ | $NEWIN(\sigma_{mB}) \equiv COMPOSE\ (\sigma_{mB}, REC\{\ \})$ |
| Component | COMPOSE: $\Sigma_{mB} \times \Sigma_{messages} \rightarrow \Sigma_{mB}$ | PRIMITIVE |
| Component | REC: $\phi \rightarrow \Sigma_{messages}$ | $REC(\{\ \}) \equiv XCLOCREC_i\{\ \}$ |
| Component | NEWOUT: $\Sigma_{IB} \rightarrow \Sigma_{messages}$ | $NEWOUT(\sigma_{IB}) \equiv (t:(CONCAT(INSYS(\sigma_{IB}), SEND(OUTSYS(\sigma_{IB}))))$ |

| TYPE | MAPPING | DEFINITION |
|---|---|---|
| Component | CONCAT: $\sum_{IB} \rightarrow \sum_{IB}$ | PRIMITIVE |
| Component | INSYS: $\sum_{IB} \rightarrow \sum_{IB}$ | $INSYS(\sigma_{IB}) \equiv INSYS(V_{IB})$ |
| Value | $INSYS_{11}$: $V_{IB} \rightarrow V_{IB}$ | PRIMITIVE |
| Component | OUTSYS: $\sum_{IB} \rightarrow \sum_{IB}$ | $OUTSYS(\sigma_{IB}) \equiv OUTSYS_{11}(V_{IB})$ |
| Value | $OUTSYS_{11}$: $V_{IB} \rightarrow V_{IB}$ | PRIMITIVE |
| *Component* | *SEND:* $\sum_{IB} \rightarrow \phi$ | $SEND(\sigma_{IB}) \equiv XCLOCSEND_i(\sigma_{IB})$ |
| State Successor | REALWORLD: $\sum_{REAL} \rightarrow \sum_{REAL}$ | $REALWORLD(\sigma_{REAL}) \equiv REALWORLD_1(\sigma_{TO_1}, \ldots \sigma_{TO_N})$ |
| Component | $REALWORLD_1$: $\prod_{i=1}^{N} \sum_{TO_i} \rightarrow \prod_{i=1}^{N} \sum_{TO_i}$ | $REALWORLD_1(\sigma_{TO_1}, \ldots, \sigma_{TO_N}) \equiv JOIN(PACKETIN(\ ),$ $TRYSEND(\sigma_{TO}, \ldots, \sigma_{TO_N})))$ |
| Component | JOIN: $\sum_{NEW} \times \prod_{i=1}^{N} \sum_{TO_i} \rightarrow \prod_{i=1}^{N} \sum_{TO_i}$ | PRIMITIVE |
| Component | PACKETIN: $\phi \rightarrow \sum_{NEW}$ | $PACKETIN \equiv CHOICE(GREC_1(\ ), \ldots, GREC_N(\ ))$ |
| Component | CHOICE: $\prod_{i=1}^{N} \sum_{FROM_i} \rightarrow \prod_{i=1}^{N} \sum_{FROM_i}$ | PRIMITIVE |
| Component | $GREC_i$: $\phi \rightarrow \sum_{FROM_i}$ | $GREC_i(\ ) \equiv XSLOCSEND_i\{\ \}$ |
| Component | TRYSEND: $\prod_{i=1}^{N} \sum_{TO_i} \rightarrow \prod_{i=1}^{N} \sum_{TO_i}$ | $TRYSEND(\sigma_{TO}, \ldots, \sigma_{TO_N}) \equiv (t{:}(GSEND(\sigma_{TO_1}), \ldots,$ $GSEND(\sigma_{TO_N})))$ |
| Component | $GSEND_1$: $\sum_{TO_1} \rightarrow \sum_{TO_i}$ | $GSEND_i(\sigma_{TO_i}) \equiv XSLOCREC_i\{\sigma_{TO_i}\}$ |

EXCHANGES

$XSLOCREC_i(\ )$
$XCLOCREC_i(\ )$
$XSLOCSEND_i(\ )$
$XCLOCREC_i(\ )$